Aspects of Embedded System Design

Mahmut Taylan Kandemir
Pennsylvania State University
University Park, PA 16802, USA
kandemir@cse.psu.edu

Acknowledgements

• Marilyn Wolfe
• Mary Jane Irwin
• Yuanrui Zhang
• Wei Zhang
• Vijay Narayanan
• Seda Ogreneci Memik
• Hakan Aydin
• Jack Davidson
Me

• My own research
  – Compiling for advanced microprocessor systems with deep memory hierarchies
  – Optimization for embedded systems (space, power, speed, reliability), with an emphasis on chip multiprocessors
  – Energy-conscious hardware and software design
  – Runtime system support for data-intensive applications

• Thus, my interests lie in
  – Quality of generated code
  – Interplay between compiler and architecture
  – Static analysis to understand program behavior
  – Runtime performance analysis and architecture design

• Visit: http://www.cse.psu.edu/~kandemir/

Outline

• Lecture #1: aspects of embedded system design
• Lecture #2: compiler directed optimizations
• Lecture #3: power and related issues
• Lecture #4: embedded OS and reliability
Aspects of embedded system design

Architectures and Platforms
- Software: threads, mapping, scheduling
- Hardware: cores, multicores, NoC

Design, Modeling, Analysis

Design Methodologies

Applications and Environments
- Characteristics, Specifications, Interactions, QoS

Embedded system constraints

Energy, Form Factor, Processing Speed, Time Constraints, Frequent Disconnectivity, Memory
Embedded system design challenges

- Strict, quantifiable goals (in most cases)
- Design space is typically very large and highly irregular (e.g., lots of metrics, lots of alternatives)
- Lack of automated tools for many steps
- Can’t simulate everything
  - We need quick, hierarchical and reasonably accurate evaluation tools
- In most cases, software and hardware should be developed concurrently

Applications

- Application characteristics and execution environment should be well studied
- What are embedded applications?
  - Consumer electronics (cell phones, PDAs, digital cameras, DVD players)
  - Household appliances (dishwashers, microwave ovens)
  - Transportation systems
  - Medical equipments (pacemakers, glucose monitors)
  - National infrastructures (power transmission networks)
Example: cell phones

- Increasing use worldwide
- Many different applications on-board
- Contain a large amount of circuitry, each of which is carefully designed to optimize performance
  - Radio frequency receiver and transmitter
  - Digital signal processor
  - A/D conversion
  - Control processor
  - Power control unit and battery

Example: automobiles as distributed embedded systems

CAN Controller Area Network
GPS Global Positioning System
GSM Global System for Mobile Communications
LIN Local Interconnect Network
MOST Media-Oriented Systems Transport
Automotive and aviation electronics

- Functions embedded in vehicles include braking assistance, active suspension, steering control, etc.
- Subject to stringent timing constraints
- Some functions are safety-critical
- Must operate in real-time
- Must fit within power budget (limited by generator)
- Must be resilient to errors
  - In both hardware and software

Other applications

- ADSL modem/router
- ABS brakes
- Artificial pacemaker
- Dishwasher
Modeling an Embedded Systems

• Modeling input and output
• Modeling interaction with physical world
  – Performance (timing) specification
  – Power specification
  – Others

• Common computation models
  – Finite state machines
  – Turing machines
  – Petri nets
  – Data flow based models
  – Control flow based models

Models of computation

• No one model of computation is good for all algorithms
• Models effect programming style
• Large systems may require different models of computation for different parts
  – Models must communicate compatibly
Data flow versus control flow

- In principle, they are equivalent
  - An execution reaching a program point can be labeled as such

- Control flow based models
  - Flow of control during execution
  - Response to events and interrupts
  - Control dependencies

- Data flow based models
  - How data is generated and consumed
  - Memory performance
  - Data dependencies

Metrics

- Performance can be measured in multiple ways
  - Latency vs. throughput
  - Average vs. worst-case vs. best-case
  - Peak vs. sustained

- Power/energy
  - Power consumption is important for heat generation
  - Energy consumption is important for battery life
  - Integrated metrics such as Energy-Delay Product (EDP)

- Cost
  - Design time vs. lifetime

- Reliability
  - Availability, dependability, performability
Hardware/software codesign

• Goals
  – Optimizing design process
  – Optimizing design

• Tasks
  – Cospecification and comodeling
  – Codesign
  – Cosynthesis, optimization, interfacing
  – Verification

Codesign flow

- Specification
- Partitioning
- SW Design
- Interface Design
- HW Design
- Integration and Test
- Evaluation
Cosynthesis

• **Software synthesis**
  – Code optimization and code generation

• **Hardware synthesis**
  – Works at different layers

• **Interface and communication synthesis**

• **Scheduling**
  – Instruction scheduling in compilers
  – Operation scheduling in architecture
  – Process/thread scheduling in OS

Software synthesis and code generation

• **Multilayer software design**
  – Operating systems
    • Embedded, real time
    • Resource allocation, scheduling

• **Compilers**
  – Code generation and optimization
  – Retargetability

• **Interfacing**
  – Between different software components
  – Between software components and hardware components
Hardware synthesis

• Automated design process that interprets the description of a desired behavior and creates hardware that implements that behavior
• Improves designer productivity
• Many commercial tools available
• Can be addressed at different levels of abstraction
  – Gate level
  – Register transfer level
  – Algorithmic level

Interface synthesis

• Automated generation of software drivers and hardware components to ease information flow between two components
• Different interfacing strategies are possible
  – Synchronous (Blocking)
  – Asynchronous
  – Self timed
  – etc
Platform-based design

• Design the platform
• Use the platform

Work partitioning

• Platform design
  – choose, characterize hardware units
  – create the system architecture
  – optimize for performance, power
• Platform-based product design
  – modify hardware architecture
  – optimize programs
Platform design challenges

- Does it satisfy basic requirements?
- Is it sufficiently customizable? And in the right ways?
- Is it cost-effective?
- How long does it take to turn a platform into a product?

Platform design methodology

- Develop an initial architecture
- Evaluate for performance, power, etc
- Evaluate customizability
- Improve platform after each use
Platform use challenges

• How do I understand the platform’s design?
• How do I modify it to suit my needs?
• How do I optimize for performance, power, etc.?

Platform use methodology

• Start with reference design, evaluate differences required for your features
• Evaluate hardware changes
• Implement hardware and software changes in parallel
• Integrate and test
Embedded architectures

- Application-specific circuits (ASICs)
- Processors
  - RISC CPUs
  - VLIWs/DSPs
  - SoCs
  - Multicores (MPSoC)
- Reconfigurable architectures
  - Fine grain (FPGAs)
  - Coarse grain
  - ASIPs, extensible processors (e.g., Tensilica’s Xtensa)

Tradeoffs

- Flexibility
  - Reconfigurable Fabrics
  - General-Purpose Processors
  - DSPs
  - Domain-Specific Accelerators
  - ASICs
- Performance
Hardware vs. software

- Hardware: functionality implemented via a custom architecture
- Software: functionality implemented in software on a programmable component
- Key differences:
  - Concurrency
    - Processors usually have one “thread of control”
      - This is changing with multicores
    - Dedicated hardware often has concurrent datapaths
  - Multiplexing
    - Software modules multiplexed with others on a processor (e.g., using an OS)
    - Hardware modules are typically mapped individually on dedicated hardware

Embedded vs. general-purpose processors

- Embedded processors may be optimized for a category of applications
  - Customization may be narrow or broad
- We may judge embedded processors using different metrics
  - Form factor
  - Memory capacity
  - Memory system performance
  - Predictability
  - Reliability
RISC processor families

- ARM: ARM7 is relatively simple, no memory management; ARM11 has memory management, other features
- MIPS: MIPS32 4K has 5-stage pipeline; 4KE family has DSP extension; 4KS is designed for security
- PowerPC: 400 series includes several embedded processors; MPD7410 is two-issue machine; 970FX has 16-stage pipeline
- Intel and AMD also have embedded product lines

ARM architecture

- As of 2007, about 98 percent of the more than one billion mobile phones sold each year use at least one ARM processor
- As of 2009, ARM processors account for approximately 90% of all embedded 32-bit RISC processors
- Used extensively in consumer electronics, including PDAs, mobile phones, digital media and music players, hand-held game consoles, and calculators
ARM Thumb

- Goal: reduce memory requirements
  - Except in the most speed-critical of embedded devices, the cost of memory is much more critical than the execution speed of the processor
- Thumb ISA consists of 16-bit instructions that act as a compact shorthand for a subset of the 32-bit instructions
- Some functions that can be accomplished in a single ARM instruction can only be simulated with a sequence of Thumb instructions
- Expansion happens at runtime (using a dedicated unit)
- Thumb code is typically slower than standard ARM code
- Tradeoff: code size vs. speed vs. power

SPMs vs caches

- Software managed on-chip memory with fast access latency and low power consumption
- Frequently used in embedded computing
  - Allows accurate latency prediction
  - Can be more power efficient than conventional caches
- Can be used along with caches
- Downside: requires software management
  - Not good for certain applications
  - Code modifications can be tricky
Very long instruction word (VLIW) processors

- The power consumption, hardware cost and complexity of the superscalar dependence analyzer and scheduler is a major problem
- VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently
- Contemporary VLIWs typically have four to sixteen functional units
- Variants of this concept are employed in many embedded processors
  - NXP's TriMedia, Analog Devices SHARC DSP, TI's C6000 family, STMicroelectronics's ST200 family
  - Tensilica's Xtensa LX2 processor (FLIX)

VLIWs

- Compiler has a bigger context from which to select co-scheduled instructions
- Compilers, however, do not have much runtime information such as cache misses
  - Scheduling is, therefore, inherently conservative
- Branch and memory prediction are more difficult
- VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are proven to be critical
Traditional VLIW architecture

• Large register file feeds multiple function units

Clustered VLIW architecture

• Register file, function units divided into clusters
**TriMedia (Mediaprocessor)**

- A VLIW mediaprocessor family from NXP Semiconductors (formerly Philips Semiconductors)
- Features many DSP and SIMD operations to efficiently process audio and video data streams
- 8 issue slots filling 45 functional units
- High level programmability (C and C++)

---

**Exploiting slacks to reduce energy consumption (1/2)**

- VLIW functional units (FUs) often are not completely utilized due to
  - lack of available instructions using all units
  - data dependences
- Consequently, there are execution slacks
  - Some operations can be executed slower without impacting the overall execution time
  - Some functional units can be completely shutdown if idle time is long enough
- Compiler can exploit these slacks to reduce energy consumption
Exploiting slacks to reduce energy consumption (2/2)

• Compiler can reschedule instructions for energy
• Goal is to have larger slacks so that
  – More opportunities to switch to low power operating modes
  – Stay longer in low power modes
• Constraint: performance impact

System-on-Chip (SoC)

• Ability to place multiple functions on a single silicon chip, cutting development cycle while increasing product performance and quality
• A typical SoC consists of:
  – One microcontroller, microprocessor or DSP core
  – Memory blocks including a selection of ROM, RAM, EEPROM and flash
  – Timing sources including oscillators and phase-locked loops
  – Peripherals including counter-timers, real-time timers and power-on reset generators
  – External interfaces such as USB and Ethernet
  – Analog-to-digital and digital-to-analog converters
  – Voltage regulators and power management circuits
Multiprocessor SoC (MPSoC)

- An SoC that employs multiple (possibly heterogeneous) processors
- MPSoC components are linked to each other by an on-chip interconnect (e.g., buses, NoC)
- Application parallelization is a big issue
  - Hand parallelization vs. compiler guided parallelization
  - Other important issues include thread-to-processing unit mapping, data access scheduling, synchronization optimizations

FPGAs

- Integrated circuit designed to be configured after manufacturing
- Contains programmable logic components called logic blocks, and a hierarchy of reconfigurable interconnects that wire blocks together
- Recent trend: combining the logic blocks and interconnects of traditional FPGAs with embedded hard and soft cores and related circuitry to have System-on-Chip
Bar() {
    int a, b, c, d, e;
    a = b + c;
    b = e * d;
    d = a / b;
    for(.....) {
        a = b + c;
        b = e * d;
        for(.....) {
            for(.....) {
                foo_a();
            }
        }
    }
    foo_b();
}

Codesign space exploration for a single function

• One can employ branch-and-bound search to find a solution with least latency under the given area constraint
• To facilitate the search, an auxiliary data structure can be used to systematically enumerate all possible combinational mappings for nodes in an LHT
Codesign space exploration for an entire program

- Handle function calls by traversing the call graph in a post order, so that the codesign exploration is performed and a solution has been found for each callee during the exploration of the caller
  - The best solution is recorded for each callee along with the area and latency costs
  - Function calls implemented outside the program are assumed to be characterized with pre-estimated costs, e.g., software libraries
Cost estimation models for LHT

• Leaf nodes
  – Software estimation – compiler based techniques
    • Area: 0
    • Latency: profiling or WCET analysis
  – Hardware estimation – modeling tools
    • an FPGA-based modeling tool
• Non-leaf nodes – based on leaf node estimations
  – Software estimation
  – Hardware estimation

Leaf node hardware estimations

[Junsub Kim et al in IEEE Transactions on Computers, 2009]
Non-leaf node software cost estimations

- Area: 0
- Latency: loop iteration count * (its children’s total latency) / degree of parallelism
  - Two types of dependencies
    - Dependencies across child loops
    - Dependencies across loop iterations
  - Communication cost between hardware and software
    - Parent-child communication
    - Child-child communication

![Loop Dependence Graph]

Iteration Dependences for loop0

Children’s latency = node1 + max (node2, (node3 + node4)) + node5

Degree of parallelism = min (parallel iterations, number of CPUs)
Non-leaf node hardware estimations

- If a non-leaf node is implemented in hardware, its entire sub-LHT is assigned to hardware
- Pipelined design is considered

Cost evaluation for the entire LHT

- Before branch-and-bound search starts, the area costs of all nodes in the mapping tree are estimated based on node types
- The search tries to calculate the total area of a mapping by accumulating the area values when traveling down along each path
- If the area constraint is met, performance cost is estimated in a bottom-up fashion on the LHT for the root, using non-leaf node cost models
Possible extensions

• Codesign for more complex hybrid embedded architectures
  – Can still use branch-and-bound search
  – Can build cost estimation models based on the characteristics of the underlying platforms

Potential research topics

• Characterization of multithreaded embedded applications
• Design and implementation of multi-FPGA systems
• Multicore DSP design and optimization
• Application optimization for heterogeneous systems
• Novel codesign strategies targeting hybrid systems
Motivation

• Increasing “software content” in embedded systems
  – Past: 10% SW, 90% HW
  – Current: 70% SW, 30% HW
  – Future: 90% SW, 10% HW
• Traditionally, software optimizations consider performance
• Additional constraints in embedded computing domain:
  – Memory space (data and instructions), energy/power behavior, error resilience
Challenges

• Retargetability
• Extensibility and adaptability
• High-quality code
• Portability
• Robustness
• Complete package

Software optimizations

• Algorithm and Application Design
• Compiler Optimizations
  – High Level (source level)
  – Low Level (code generation)
• Operating System Control
  – Includes runtime optimizations as well
Compiler optimizations

- Classical optimizations
  - Target high end machines
  - Typically do not have power constrains or memory space limitations
- Embedded system specific optimizations
  - Performance
  - Memory space
  - Power (will be covered later) – Lecture #3
  - Reliability (will be covered later) – Lecture #4

Classical optimizations

- Back-end optimizations
  - Instruction selection, resource allocation, scheduling, etc.
- High level optimizations
  - Loop optimizations
    - Linear transformations, tiling, loop merging/distribution
  - Data layout optimizations
    - Changing memory layout of data, dimension re-indexing
- Parallelism related optimizations
**Exercise #1**

```plaintext
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        a[j][i] = b[j][i]+1;
```

**Exercise #2**

```plaintext
for (i=1;i<N;i++)
    for (j=0;j<N-1;j++)
        a[j][i] = a[j+1][i-1]+1;
```
Exercise #3

for (i=0; i<N; i++)
for (j=0; j<N; j++)
    ... = a[i] + b[j];

for (jj=0; jj<N; jj=q)
for (i=0; i<N; i++)
for (jjj=jj; j jj+jj+q)
    ... = a[i] + b[jjj];

Loop transformations

- Data dependencies may be within or between loop iterations
  - The latter is more problematic to handle
- Any code transformation should preserve data dependencies

[High-Performance Embedded Computing, M. Wolf]
Types of loop transformations

• Loop permutation changes order of loops
  – Loop interchange is a specific instance
• Loop tiling helps to exploit temporal reuse in outer loop iterations
• Loop unrolling copies the loop body
• Loop fission creates separate loops for statements in the loop body
• Loop fusion combines bodies of neighboring loops

• These optimizations can be used for maximizing cache performance and/or improving parallelism

Example: loop permutation and fusion

Original loop nest

```plaintext
for (i=0; i<N; i++)
  x[i] = a[i] * b[i];
for (i=0; i<N; i++)
  y[i] = c[i] * a[i];
```

After loop permutation

```plaintext
for (j=0; j<M; j++)
  for (i=0; i<N; i++)
    x[i][j] = a[i][j] * b[j];
for (j=0; j<M; j++)
  y[i][j] = c[i][j] * b[j];
```

After loop fusion

```plaintext
for (i=0; i<N; i++)
  { x[i] = a[i] * b[i];
  y[i] = c[i] * a[i]; }
```
Embedded system specific optimizations

• Instruction space optimizations
  – Offset assignment
  – Code positioning
• Data space minimization
  – Memory space reuse
  – Compiler directed data compression
• Compiler directed memory hierarchy design
• WCET estimation
• Power optimization (more on this later)
• Reliability optimizations (more on this later)

Minimizing code space

• Smart storage assignment
  – Address assignment to variables
  – Effective use auto-increment/decrement addressing modes in DPSs
• Code compression
  – Software Approach
    • Common sequences are extracted and placed in a directory
    • Instances of these sequences are replaced by mini-subroutine calls (automatic subroutine creation)
  – Hardware Approach
    • New instructions are defined
    • Flexible dictionary structures with architectural support
Storage assignment

- Many DSPs have limited addressing modes (e.g., TMS320C55x)
- They use address registers to access memory
- No complex indexing modes; instead they have auto-increment/decrement modes
- Compiler support is needed to minimize the number of explicit assignments to address registers
- Address instructions can constitute up to 55% of total program bits

Use of address registers

\[ \text{.....x.....y.....} \quad \rightarrow \quad \begin{align*}
\text{add r4, M(AR)} \\
\text{AR}=\text{AR}+q \\
\text{sub r8, M(AR)}
\end{align*} \]

\[ \text{.....xy.....} \quad \rightarrow \quad \begin{align*}
\text{add r4, M(AR+)} \\
\text{sub r8, M(AR)}
\end{align*} \]
Offset assignment problem

- Simple offset assignment
  - One address register
  - Works with +1/-1 offset
  - Operates on a single basic block at a time
- General offset assignment
  - Multiple registers, larger offset values, inter-basic block analysis, etc

Simple offset assignment

\[
\begin{align*}
c &= c + d + f \\
a &= h - c \\
b &= b + e \\
c &= g - b \\
a &= a - c
\end{align*}
\]

Based on order of declaration:
\[a,b,c,d,e,f,g,h\]

Based on order of first use:
\[c,d,f,h,a,b,e,g\]

Based on compiler analysis:
\[a,c,h,g,b,e,f,d\]
Simple offset assignment

- Access sequence for “z=x op y” is “xyz”
- Access sequence for an ordered set of operations is the concatenated access sequences for each operation in the appropriate order
- Cost of assignment: the number of adjacent accesses of variables that are not assigned to consecutive places
- Can be formulated as Graph Covering [S. Liao, MIT]

Access graph

c,d,f,c,h,c,a,b,e,b,g,b,c,a,c,a
General offset assignment

- General offset assignment (GOA) is the problem of optimizing the memory layout when there are multiple address registers
- Solutions break access sequence into multiple sequences and then solve the SOA problem
General offset assignment

• **Straightforward approach**
  – Fix the access sequence
  – Form subsequences
  – Solve a separate SOA for each subsequence

General offset assignment

• **Three approaches for assigning variables to address registers**
  – Greedy algorithm [Luepers & David 1998]
  – Heuristic approach that iteratively partitions variables [Sugino et. al 1996]
  – Coalescing nodes in the access graph [Zuang et. al 2003]
General offset assignment

- Access sequence: 
  a d b e c f b e c f a d
- Sequence 1: a b c b c a
  Sequence 2: d e f e f d
- Memory layout

An alternate approach

- Modify code sequence instead of memory layout
- Problems:
  - Data dependences
  - Performance issues (i.e., conflict with the code scheduler)

- Extension to array computations
Memory placement of program code

- Goal is to place code carefully in memory to improve cache performance
- Possible cache conflicts may be determined using addresses; interesting conflicts are determined through analysis
- May require program padding

McFarling

- Analyzed program structure, trace information
- Annotated program with loop execution count, basic block size, procedure call frequency
- Walked through program to propagate labels, group code based on labels, place code groups to minimize interference
Pettis and Hansen

- Profiled programs using gprof
- Put caller and callee close together in the program, increasing the chance they would be on the same page
- Ordered procedures using call graph, weighted by number of invocations, merging highly-weighted edges
- Optimized if-then-else code to take advantage of the processor’s branch prediction mechanism
- Identified basic blocks that were not executed by given input data; moved to separate processes to improve cache behavior

Reducing data memory area (1/4)

```plaintext
for(i=0;i<N;i++)
  ... = c[i]
for(i=0;i<N;i++)
  b[i] = a[i]
for(i=0;i<N;i++)
  ... = c[i]
for(i=0;i<N;i++)
  c[i] = a[i]
```

Reusing the same memory space
- Can reduce capacity misses
- Can lead to smaller memory in embedded design

last use
Reducing data memory area (2/4)

```plaintext
for(i=1; i<N; i++)
    for(i=0; j<N-1; j++)
        a[i][j] = ...
        ...
          = a[i-1][j+1]
```

- **Iteration** \( (i, j)^T \)  →  write into \( a[i][j] \)
- **Iteration** \( (i+1, j-1)^T \)  →  read from \( a[i][j] \)

After \( (i+1, j-1)^T \), memory location of \( a[i][j] \) can be used for storing another element

Reducing data memory area (3/4)

**Strategies:**
- Inter-variable
  - Use \( a[] \) instead of \( b[] \)
- Intra-variable
  - Use \( a[3] \) instead of \( a[44] \)

Re-writing the code is a big issue!
Reducing data memory area (4/4)

• One method:
  – Divide data arrays into (disjoint) chunks
  – Create a node for each chunk
  – If two chunks have overlapping lifetimes, put an edge between them
  – The result is an **Interference Graph (IG)**
  – Color the graph (remember this?)

Example: NUFFT

• Variables with the same color have non-overlapping lifetimes
• Variables *cone*, *rl’*, *rll*, *Angll* and *l* can share the same storage.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Original Storage</th>
<th>Optimized Storage</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test.m</td>
<td>28751</td>
<td>26541</td>
<td>2210</td>
</tr>
<tr>
<td>fmm.m</td>
<td>1180</td>
<td>816</td>
<td>364</td>
</tr>
<tr>
<td>cluster_fmm.m</td>
<td>15953</td>
<td>12257</td>
<td>3696</td>
</tr>
<tr>
<td>trans_fmm.m</td>
<td>22125</td>
<td>16885</td>
<td>5240</td>
</tr>
</tbody>
</table>
Memory occupancy

- **MMO: Maximum memory occupancy**
  - captures the amount of memory that needs to be allocated for the application
- **AMO: Average memory occupancy**
  - important in a multiprogramming based environment where multiple apps compete for the same space
- **Drops in the curve indicate recycling**

![Graph showing memory occupancy over time]

Compiler-guided data compression for reducing memory consumption

- **Goal**
  - Keep the data block in the on-chip memory even if the reuse distance is large
    - Compress data blocks with large inter-access times
    - When next request comes, decompress the data block and forward it to the requester
- **Advantages**
  - Data is kept on-chip
  - Less memory occupation
- **Drawback**
  - **Decompression**
    - We should not compress the data block if its reuse distance is short
    - Reduced on-chip space utilization
Challenges

- Which data blocks should be compressed and decompressed
- Order of compressions and decompressions?
- Data sharing across the processors must be accounted for
  - Decisions should be made based on global data access patterns
- Original execution cycle count should not increase excessively
  - Need to be careful about the critical path of execution
  - Complex compression/decompression algorithms should be avoided

Additional compiler tasks

- Divide data arrays into tiles
- Compiler produces two (helper) threads
  - Compressor and decompressor
- Compressor implements data tile compressions
- Decompressor implements data tile decompressions
- Scheduling these threads is the key
  - In particular, decompressions can come in the critical path of execution
  - Decompressions can be overlapped with compressions (pre-decompression)
- Tradeoffs between memory space consumption and performance
Experimental evaluation

- Memory space occupancy for a stencil code
- 20 epochs with the same number of execution cycles

- Memory occupancy of BASE continuously increases
- The best space savings are achieved with the CD+LF version
  - Combines data compression and dead block recycling

Compiler-directed memory hierarchy design

- Software-managed hierarchies can be preferable against hardware counterparts in embedded computing
  - Usually the best choice for embedded systems that execute a single application
    - Customized memory hierarchy that suits the needs of the application
  - Data movements are handled by software
    - Energy efficient (no dynamic lookups)
- Can be applied to single core as well as multicore design
Data reuse and data locality

• Reuse
  – Multiple use of the same data
  – A characteristic of application; determined by access pattern
  – Independent of memory architecture

• Locality
  – Catching the reused data in fast memory
  – Depends on fast memory capacity, management policy, and application access pattern

Objective of locality optimization

• Converting reuse into locality
  – Loop optimizations
  – Data layout transformations
  – or both…

• Requires program analysis, checking data dependences, capturing data reuse, and transforming code
Representation

• Each array reference represented by

   \[ HI + c \] (an affine function)

   • \( I \): iteration vector (loop indices)
   • \( H \): access (reference) matrix
   • \( c \): offset vector

• Example:

   
   \[
   \begin{align*}
   &\text{for } i = 1, N \\
   &\text{for } j = 1, N \\
   &\ldots \ U[i+1][j+2] \\
   \end{align*}
   \]

   \[
   H= \begin{pmatrix}
   1 & 0 \\
   0 & 1 \\
   \end{pmatrix}, \quad c= \begin{pmatrix}
   1 \\
   2 \\
   \end{pmatrix}
   \]

Data reuse vectors

• Data reuse analysis
  – \( HI+c = HJ+c \) \( \rightarrow \) \( H(I-J)=0, \ r=I-J \)
  – \( r \): temporal reuse vector
  – \( H'I+c = H'J+c \) \( \rightarrow \) \( H'(I-J)=0, \ r'=I-J \)
  – \( r' \): spatial reuse vector

• Example
  – In a nest with loops i, j, k
  – \( V[i][k] \) has temporal reuse in j loop
  – Successive iterations of j loop access the same array element
What does reuse vector say?

- If we have two levels of memory
  - Fast memory (small, energy-efficient)
  - Slow memory (large, power-hungry)
- We can reuse vectors to guide data transfers
  - $V[i][k]$ has temporal reuse in $j$ loop
  - Bring $V[i][1..N]$ into fast memory before $j$ loop
  - Write it back to slow memory after $j$ loop
  - Very good (expected) energy/performance behavior during execution of $j$ loop

Playing with reuse vectors

- If $r$ is a reuse vector and $T$ is a loop transformation, $Tr$ is the new reuse vector
- Optimization problem
  - Use loop transformations for modifying reuse vector(s)
  - Select a $T$ such that $Tr$ is in a desired form
  - This can be used for locality optimization, or customized memory hierarchy exploration
Types of reuse vectors

- $r = [0, 0, 0, 1, 0, 0]^T$
  - Reuse in a single loop: single fast memory
- $r = [0, 0, 1, 0, 1, 0]^T$
  - Reuse in multiple levels: multi-level memory
- Loop transformations can convert one type of reuse vector into another

Types of transformations

- $[0, 0, 0, 1, 0, 0]^T \rightarrow [0, 0, 0, 0, 1, 0]^T$
  - Changes reuse level
  - Affects capacity of fast memory
- $[0, 0, 0, 0, 1, 0]^T \rightarrow [0, 0, 0, 1, 0, 1]^T$
  - Changes the number of reuse levels
  - Affects the number of memories in the hierarchy
Exploiting temporal locality in memory design

Array Index Values

Can be kept in faster memory

Reuse Region

Time

Exploiting temporal locality in memory design (multiple levels)

Array Index Values

in the next fastest memory

in the fastest memory

Time
Memory trees

Individual memory trees

Combined memory tree

[Catthoor et al 98]

Multi-level on-chip memory hierarchy design

- Alternatives
  (a) pure shared memory
  (b) pure private memory
  (c) memory space across a single level (not a hierarchy)
  (d) hierarchy based design
- In the most general case
  - A “hybrid” architecture
  - Each processor can see a different memory hierarchy image
Overview of a profiler/compiler directed approach

- Input program is divided into phases and profiled/statically analyzed
- The profile/analysis data data gives
  - Data blocks accessed by each proc
  - An estimation of the number of accesses to each data block by each processor
- This information is then passed to a solver, which
  - Determines the locations of data blocks for each phase
  - Preferred on-chip memory hierarchy
- This information is passed to the compiler
  - Modifies the application code accordingly to insert explicit data transfer calls

Varieties of performance metrics

- **Worst-case execution time (WCET):**
  - Factor in meeting deadlines
- **Average-case execution time:**
  - Factor in identifying problematic spots
- **Best-case execution time (BCET):**
  - Factor in coming up with more realistic bounds (when used along with WCET)
Performance analysis techniques

• Simulation
  – Not exhaustive
  – Cycle-accurate CPU models are often available

• WCET analysis
  – Formal method; may make use of some simulation techniques
  – Bounds execution time but hides some details

Sources of variations

• Inputs
• Runtime program paths
• Execution-specific timing
  – Caches
  – Pipelines
  – Branch Prediction
  – Speculation
  – OOO Execution
How to Estimate WCET Safely and Accurately

- Estimated WCET must be:
  - Safe
  - Tight
- Industry practice: measurement + safety margin
- Measurement based approaches are *not* safe
- Another promising approach: static WCET analysis

Potential research topics

- Compiler directed memory hierarchy design
- Scratch-pad optimizations for multicore architectures
- WCET analysis for cache based multicores
- Global code analysis for tighter WCET and BCET
- Code optimizations for multicore embedded systems
Last class

- Instruction space optimizations
- Data space optimizations
- Reuse vectors and their importance

Representation

- Each array reference represented by $HI + c$ (an affine function)
  - $I$: iteration vector (loop indices)
  - $H$: access (reference) matrix
  - $c$: offset vector

- Example:
  \[
  I = \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}, \quad H = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 2 \end{pmatrix}
  \]

  for $i_1 = 1, N$
  for $i_2 = 1, N$

  ... $U[i_1+1][i_2+2]...$
Data reuse vectors

- **Data reuse analysis**
  - $HI+c = HJ+c \rightarrow H(I-J)=0, r=I-J$
  - $r$: temporal reuse vector
  - $H'I+c = H'J+c \rightarrow H'(I-J)=0, r'=I-J$
  - $r'$: spatial reuse vector

- **Example**
  - In a nest with loops $i$, $j$, $k$
  - $V[i][k]$ has temporal reuse in $j$ loop
  - Successive iterations of $j$ loop access the same array element

Example calculation

```plaintext
for(i=1;i<N;i++)
    for(i=0;j<N-1;j++)
        a[i][j] = ....
        .... = a[i-1][j+1]
```

Iteration $(i \ j)^T \rightarrow$ write into $a[i][j]$

Iteration $(i+1 \ j-1)^T \rightarrow$ read from $a[i][j]$

$r = (i+1 \ j-1)^T - (i \ j)^T = (1 \ -1)^T$
Ranking reuse vectors

• Which one is good?
  – (1 0 0)^T or (0 1 0)^T or (0 0 1)^T

• Can we change a reuse vector?
  – E.g., how can we go from (1 0)^T to (0 1)^T?

• Answer: using loop transformations
  – If \( I = (i_1, i_2)^T \) is the original iteration vector and we apply a linear transformation represented by \( T \), it is transformed to \( I' = TI \)
  – And, \( HIo \) is transformed to \( HT^{-1}I' + o \)  (Why?)
  – And, \( r \) is transformed to \( r' = Tr \)  (Why?)
  – Our goal is to find \( T \) such that \( r' \) is in a desired form

Potential uses of r

• Data locality optimization
• Memory space minimization
• Memory hierarchy design
  – \( r = (0 0 0 1 0 0)^T \)
    • Reuse in a single loop: single fast memory
  – \( r = (0 0 1 0 1 0)^T \)
    • Reuse in multiple levels: multi-level memory
Power and Related Issues

Mahmut Taylan Kandemir
Pennsylvania State University
University Park, PA 16802, USA
kandemir@cse.psu.edu

Motivation

• Power and energy are first-class parameters in many embedded designs and execution environments
• Memory energy is particularly problematic
• Support for low-power operating modes motivates for energy-aware optimizations
• Code-level optimizations and data layout changes may increase savings

• The goal is to save energy/power without significantly impacting performance
Heat becoming an unmanageable problem!

Power and energy design space: a hardware perspective

<table>
<thead>
<tr>
<th></th>
<th>Constant Throughput/Latency</th>
<th>Variable Throughput/Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy</td>
<td>Design Time</td>
<td>Non-active Modules</td>
</tr>
<tr>
<td>Active</td>
<td>Logic Design</td>
<td>Clock Gating</td>
</tr>
<tr>
<td></td>
<td>Reduced $V_{dd}$ Sizing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Multi-$V_{dd}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leakage</td>
<td>+ Multi-$V_T$</td>
<td>Sleep Transistors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multi-$V_{dd}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Variable $V_T$</td>
</tr>
</tbody>
</table>
Low-power operating modes

- Each mode is characterized by
  - Per access (or per cycle) energy consumption
  - Resynchronization cost
- Selecting the best mode involves a tradeoff between performance and energy consumption
- Compiler/OS/user must decide what to shut down (or switch to a low power mode) and when to do so. Two cases to consider:
  - If we shut down a component and that component is needed again quickly, an annoying delay might occur when it is restarted
  - On the other hand, if we wait too long to shut down a component, energy may be wasted unnecessarily
- The objective then is to carefully develop a strategy that decides about what to shut down and when

Proactive approach

- A Proactive Strategy is about predicting the future
- A compiler can analyze data access patterns and map this information to component use
- An OS can look at past access pattern information and predict what would happen in the future
- A combination can work by letting the compiler pass application-wide information to the OS; the latter then can make global (workload wide) decisions considering inputs from all co-runners
Activity clustering

• A generic technique
• Goal
  – Perform accesses in a short period of time
  – Increase lengths of idle periods
• Result
  – Can switch to a more aggressive low power mode
  – Can stay in a low power mode longer
  – Can minimize state transitions (reliability angle)

Sample energy consumptions and resynchronization costs (memory)

<table>
<thead>
<tr>
<th>Operating Mode</th>
<th>Energy Consumption</th>
<th>Resynch Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE</td>
<td>3.570 nJ</td>
<td>0</td>
</tr>
<tr>
<td>STANDBY</td>
<td>0.830 nJ</td>
<td>20</td>
</tr>
<tr>
<td>NAP</td>
<td>0.320 nJ</td>
<td>300</td>
</tr>
<tr>
<td>POWERDOWN</td>
<td>0.005 nJ</td>
<td>9000</td>
</tr>
</tbody>
</table>
Compiler-directed operating mode selection

- Estimating idle periods
  - Easy for array-based applications
    - Can be conservative in others
  - Requires mapping from cycles to loop iterations
- Preactivating memory banks
  - Not to incur re-synch costs
  - Implementation similar to compiler-directed prefetching

Iterations and access patterns

- Each loop nest can be represented using a set of iterations (iteration space):
  $$\{l\} = \{(1,1,1),(1,1,2),\ldots,(1,1,n), (1,2,1),(1,2,2),\ldots,(n,n,n)\}$$
  - An access pattern is a function from iteration space to data space
    $$\{l\} \rightarrow \{D\}$$
  - Data space (array space) consists of all array indices
    $$\{D\} = \{(1,1),(1,2),\ldots,(1,n),(2,1),\ldots,(m,m)\}$$
Classification and optimization: a data centric strategy

- \{I\}_\text{all} : all iterations in a given nest
- Assuming two memory banks, \{I\}_\text{all} can be written as
  - \{I\}_\text{all} = \{I\}_01 \cup \{I\}_10 \cup \{I\}_11
- Instead of executing iterations in \{I\}_\text{all} in their original order, we can execute
  - \{I\}_01 \rightarrow \{I\}_11 \rightarrow \{I\}_10
- Not easy and not always possible

Pattern classification

- A two-bank memory system
- Three different access patterns
  - \{I\}_01 = \{I2, I3\}
  - \{I\}_10 = \{I1, I4\}
  - \{I\}_11 = \{I0, I5\}
- Each of them is a class
Pattern optimization

- Going from \{I\}_{all} to \{I\}_01 \rightarrow \{I\}_11 \rightarrow \{I\}_10
- Converting
  \begin{align*}
  I0 &\rightarrow I1 \\
  I1 &\rightarrow I2 \\
  I2 &\rightarrow I3 \\
  I3 &\rightarrow I4 \\
  I4 &\rightarrow I5 \\
  I5 &\rightarrow I1 \\
  I1 &\rightarrow I4
  \end{align*}

Original access pattern

<table>
<thead>
<tr>
<th></th>
<th>B0</th>
<th>B1</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Optimized access pattern

<table>
<thead>
<tr>
<th></th>
<th>B0</th>
<th>B1</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problems and solutions

- How to enumerate iterations in a given class?
  - A polyhedral tool can be used
  - Example question to a polyhedral tool:
    - *Give us all iterations that access only the first bank*
- How to handle data dependences?
  - Capture them in a graph and traverse the graph using directed edges
**Class-level dependence graph (CLDG)**

- Each node is a class
- An edge from node \( I^i \) to node \( I^j \) indicates that there is a data dependence from an iteration in \( I^i \) to an iteration in \( I^j \)
- Traversing this graph using edges gives a legal execution order
- There might be more than one legal order
- Select the one that maximizes duration of idleness

**Optimization strategy**

- Associate a **bit vector** with each class
  - e.g., 10001 indicates that only the first and the last banks (of a five-bank system) are accessed
  - All iterations in a class are executed successively
- In going from one node to another, we need to minimize the bit transitions
  - Intra-class vs inter-class
- A classical scheduling problem (energy-aware scheduling)
- Lots of algorithms in literature
  - e.g., list scheduling
Transformations

• **Node Merging:**
  – Too many nodes in CLDG
  – Reduces the number of nodes and the effectiveness of our strategy
  – Sometimes it is a Must-Do optimization (cyclic CLDGs)

• **Node Splitting:**
  – To break a large cyclic dependence chain

---

**Percentage energy savings**

![Bar chart showing percentage energy savings for various benchmarks with and without cache and locality.]
Heap and garbage collector

• Heap
  – All Java objects created by the application are allocated from the heap

• Garbage Collector
  – Reclaims heap memory occupied by objects no longer needed by the application

• How Does M&S Garbage Collector works?
  – Mark – find out garbage
  – Sweep – free the memory occupied by garbage
  – Compact – combine fragments (optional)

Mark / sweep / compaction

Before GC

After Mark

After Sweep

After Compact

Live  Garbage  Unknown  Free
1. Each bank can be turned on/off independently
2. When a bank is turned off, it consumes very little energy and data in it is lost
3. After GC, turn off banks not containing live objects
Impact of GC-controlled bank turnover

On average, 30% heap energy is saved

Impact of GC frequency

Earlier detection of banks containing only garbage
- More memory accesses
- More CPU operations
- Longer execution time

More frequent GC
Adapting GC frequency automatically

- Different applications may have different optimal frequencies
- Even different phases of a single application can have different optimal frequencies
- Adaptive
  - Invoke GC whenever there is an opportunity to turn off a bank
  - Should not invoke GC unless a certain number of objects have become garbage

\[
\text{Garbage created during } [t_1, t_2] \\
k_2 = \frac{h_2 - a_2}{t_2 - t_1}
\]

\[
\text{Objects created during } [t_1, t_2] \\
k_1 = \frac{h_2 - a_1}{t_2 - t_1}
\]

GC is invoked at \( t_1, t_2 \)
**GC invocation: case 1**

If, after GC, at least 2 banks can be turned off, the garbage collector should be invoked

\[ b(h_i+s_i)-b(h_i+s_i-k_2(t_i-t_2)) \geq 2 \]

- \( b(x) \): the number of banks for size \( x \)
- \( h_i \): the total object size in the heap
- \( s_i \): the size of the object to be allocated
- \( t_2 \): time of previous collection

**GC invocation: case 2**

If only 1 bank can be turned off, then the space GC creates must allow the application to run an interval no shorter than \( L \)

\[ b(h_i+s_i)-b(h_i+s_i-k_2(t_i-t_2)) = 1 \]

and

\[ b(h_i+s_i-k_2(t_i-t_2))B-(h_i+s_i-k_2(t_i-t_2)) \geq k_1L \]

- \( B \): the size of a bank

**Purpose:** to avoid thrashing
GC invocation: case 2 (thrashing)

GC is invoked at T1, T2 and T3

Impact of adaptive GC invocation

<table>
<thead>
<tr>
<th>Heap Energy</th>
<th>10</th>
<th>40</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>base</th>
<th>adpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypto</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kvideo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kwml</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduler</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Object lifetime stages

- **Alive**
  - Will be used in the future
- **Last used**
  - Will never be used in the future, but still reachable from the roots
- **Dead**
  - Unreachable from the roots, but not reclaimed
- **Free (Collected)**
  - Reclaimed by Garbage Collector
  - Returned to the pool of free memory
### Cache line stages

<table>
<thead>
<tr>
<th>Cache-line Stage</th>
<th>Contains bytes from objects in lifetime stage:</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alive</td>
<td>Alive</td>
<td>Last used</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>Possible</td>
</tr>
<tr>
<td>Last Used</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dead</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Free</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

#### Cacheline Stages Distribution Over Runtime (1)

Benchmark: Webviewer

- **FREE**
- **DEAD**
- **LASTUSED**
- **ALIVE**
- **INVALID**
• **Last class**
  – Reuse vectors and their importance
  – Activity clustering

• **Today**
  – Compiler directed power optimizations
  – OS based approaches
  – Software based approaches to reliability problems
Power Mode Control

- Transition to a low power mode after a period of time
- Reactive vs proactive

Multicore sensors and controls

- Power/perf/fault "sensors"
  - current & temp
  - hw counters
- Turn off idle and faulty PEs
- Apply DVFS
  - Global DVFS
  - Local DVFS
    - Static (set V&F per core)
    - Dynamic (adjust V&F in response to core idleness)
Exploiting load imbalance

```
for i: 1..1024
for j: i..1024
...X[i][j]...
```

- Compiler does both loop iteration assignment and voltage assignment

- **Goal:** Save as much energy as possible without incurring an excessive performance penalty

**Block Assignment**

**Voltage Assignment**

\( V_0 > V_1 > V_2 > V_3 \)

Necessary compiler support

- Loop nest parallelization
  - Parallelize the outermost loop that carries no data dependence
  - This helps to minimize synchronization cost

- **Load estimation**
  - Iteration count estimation
  - Per-iteration cost estimation

- **Voltage and frequency assignment**
  - Processor with largest workload gets highest V & fastest F
  - For the remaining processors, choose the lowest V & slowest F that do not cause the execution time to exceed that of the processor with the largest workload

**For loop body cost of L**

- P0: 256*1024*L
- P1: 256*(1024-257+1)*L
- P2: 256*(1024-513+1)*L
- P3: 256*(1024-769+1)*L
Potential energy savings

Better savings with more PEs (more load imbalance)!

Large difference between 4 and 8 voltage levels!

Overview of compiler-directed link allocation

Input Code: Determine a route for each message

Goal: Determine a route for each message

Output Code

Building Connection Interference Graph

Determine Routes for Connections

Determining Contents of Routing Tables

Building Connection Interference Graph

Determining Routes for Connections

Determining Contents of Routing Tables
Example: X-Y routing

Example: compiler-directed, power-aware routing
**OS-based energy management**

- OS has global view of the system
- Information about actual physical frame allocation
- OS can determine points, during the execution of an application, where banks would remain idle, so they can be transitioned to low power modes
Scheduler-based approach

- Transitions the banks at context switch time
- We expect the same banks will be used in the next quantum
- We need to keep track of the banks that are used by the application per time quantum
- **Bank Usage Table (BUT)**

Bank usage table (BUT)

<table>
<thead>
<tr>
<th>Process</th>
<th>Bank Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B1</td>
</tr>
<tr>
<td>P1</td>
<td>X</td>
</tr>
<tr>
<td>P2</td>
<td>X</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Pm</td>
<td>X</td>
</tr>
</tbody>
</table>

- One row per application/task
- An X means a bank was used in the previous quantum
Integrated approach

Hardware based (reactive) + OS based (proactive)

Potential research topics

- Interfacing hardware based, compiler based, and OS based power saving strategies
- Power optimizations for multicore DSPs
- Compiling for power in the context of multicores
  - Instead of maximizing performance, explore power-performance tradeoffs
- Thermal aware computation
  - Application mapping, scheduling
  - Sensor placement
Embedded OS and Reliability

Mahmut Taylan Kandemir
Pennsylvania State University
University Park, PA 16802, USA
kandemir@cse.psu.edu

Resiliency/reliability issues in multicores

- Run away leakage on idle cores
- Thermal emergencies
- Transient errors
- Wearouts
Reliability, power, performance

How to allocate cores and map threads to handle run-time availability changes?

Program Execution

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>(11,11)</td>
</tr>
<tr>
<td>9</td>
<td>(16,9)</td>
</tr>
<tr>
<td>8</td>
<td>(14,14)</td>
</tr>
<tr>
<td>8</td>
<td>(16,14)</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Best EDP Choices for FFT

- Scenario 1: (16,14) 20% reduction
- Scenario 2: (11,11) 52% reduction
- Scenario 3: (16,9) 56% reduction

How to automatically decide the ideal configuration?
FFT

- FFT uses a 3D FFT and then the inverse 3D FFT
- Computations inside each iteration are similar

Helper thread based adaptation

- Helper thread's recipe
  - Collect performance/energy statistics as the execution progresses
  - Calculate EDP values
  - Use curve fitting to determine the best configuration to minimize EDP
Adapting the number of cores

- Number of available cores drops from 16 to 14

Adapting both threads and cores
Adapting to multiple changes

OS-guided core partitioning
Core partitioning example

EDPG:
LU 19.1% and MG 10.1%
i.e.,
W-EDGP 15%

Results with two application workloads
Results with three application workloads

Multicore architecture with variations

- From Bin 4 to Bin 1
  - Clock frequency degrades by 5%, 10% and 15%

- From Bin 1 to Bin 4
  - Subthreshold leakage power ($I_{\text{leak}} V_{dd}$) increases by 20%, 40% and 60%

[Humenay, Tarjan, Skadron; DATE’07]
Different mappings for a single-threaded application

- Results normalized to Bin 1 (the slowest, but least leaky Bin)

Which bin to run in is application dependent

Different mappings for a multi-threaded application (4 threads)

- One thread per core and all threads of an application run in the same Bin

For most applications, Bin 1 (the slowest, least leaky) gives the best EDP
Allowing bin frequencies to change

- Allow Bins to run at different frequencies (i.e., downgrade a Bin from a higher frequency to a lower frequency)

Mapping at runtime

- Non-preemptive
  - Only use idle cores
  - Alternative schemes
    - MOST: 2 cores at \( f_2 \)
    - FAST: 1 core at \( f_4 \)
    - SLOW: 1 core at \( f_1 \)

- Preemptive
  - Interrupt running applications and optimize globally
Compiler directed network-on-chip reliability enhancement

- **Idea:** Improve on-chip communication reliability by duplicating messages traveling over the network
- **Performance Constraint:** A duplicate message should use a different set of links as much as possible
- **Power Concern:** Reuse communication links across the different phases of the program to maximize link shutdown opportunities

Automatic code replication for reliability

- Not all processors are used to execute a loop
- Exploit idle processors
  - Switch off to save power
  - Execute replicated computation to increase reliability

\[
\text{Checksum} = ?
\]
Energy-delay-fallibility product

- Power, performance, and reliability tradeoffs

<table>
<thead>
<tr>
<th>Energy efficiency</th>
<th>Performance</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-power mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop replication</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Reliability: Percentage of replicated iterations
- Fallibility: Inverse of Reliability
- EDF = Energy*Time*Fallibility
  - Measures the tradeoffs between power, performance, and reliability

Energy-delay-fallibility product

Normalized EDF

Amount of Duplication

Normalized EDF vs. Amount of Duplication for different duplication methods.
Memory Conscious Loop Duplication

- Memory are doubled in full array duplication
- Utilize inactive (dead) array elements to store duplicates

![Diagram of Memory Conscious Loop Duplication]

```
int A(N);  
for i=0,N-2 {  
    A(i)=A(i)+a;  
    A'(i)=A'(i)+a;  
    if A(i)!=A'(i+1)  
        error();  
}
```

Original program

```
int A(N), A'(N);  
for i=0,N-2 {  
    A(i+1)=A(i)+a;  
    A'(i+1)=A'(i)+a;  
    if A(i+1)!=A'(i+1)  
        error();  
}
```

Full duplication

```
int A(N+2);  
for i=0,N-2 {  
    A(i+1)=A(i)+a;  
    A(i+3)=A(i+2)+a;  
    if A(i+1)!=A(i+3)  
        error();  
}
```

Memory-conscious duplication
Memory conscious loop duplication

Checkum reuse

Original Loop

Checksum Comparison

Checksum Generation

Checksum Calculation

Checksum Generation

The overhead brought by checksum generation could be removed by checksum reuse
### Intra-procedural checksum reuse

- Procedure ProcA
  - real P(100,100)
  - call ProcC(P);
- Procedure ProcB
  - real Q(100,100)
  - call ProcC(Q);

### Inter-procedural checksum reuse

Three Options:
- Do nothing
- Choose one region shape
- Procedure cloning

### Runtime integrity checking for inter-object connections

- Each C1 object points to at least two C2 objects
- All C3 objects are connected together
- Each C4 is pointed by at least three C5 objects
- Each C1 object points to a C2 object that also points back to this C1 object
- Each C2 object is pointed by at least one C5 array object or two C6 objects
## Runtime integrity checking for inter-object connections

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L ::= \forall \ C \ B$</td>
<td>For each object of type $C$, the rule body $B$ must be satisfied.</td>
</tr>
<tr>
<td>$B ::= S \mid S \lor B$</td>
<td>Each rule body consists of the disjunction of subrules.</td>
</tr>
<tr>
<td>$S ::= \Box$</td>
<td></td>
</tr>
<tr>
<td>$R ::= \exists R \ N \ast C$</td>
<td>All objects of the checking class are weakly connected.</td>
</tr>
<tr>
<td>$C ::= C_1, C_2$</td>
<td>There exist at least $N$ objects of type $C$ that have relation $R$ with the checking object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C[]$</td>
<td>$primary[]$</td>
<td>$N ::= &lt;PosInt&gt;$</td>
</tr>
<tr>
<td>Instance class.</td>
<td>Reference array class.</td>
<td>Primary array class.</td>
</tr>
</tbody>
</table>

$N$ represents positive integer number.

### Diagrams

- $\forall C_1 \ \exists \rightarrow 2 \ast C_2$
- $\forall C_3 \ \Box$
- $\forall C_4 \ \exists \leftarrow 3 \ast C_5$
- $\forall C_1 \ \exists \rightarrow 1 \ast C_6$
- $\forall C_2 \ \exists \leftarrow 1 \ast C_5[]$ or $\exists \leftarrow 2 \ast C_6$
### Potential research topics

- Quantifying vulnerability of embedded applications
- Compiler support for reliability
- Application/domain specific resiliency measures
- Reliability-power-performance tradeoffs
- Reactive versus proactive strategies