Chapter 3-1: Programs

Soo-Ik Chae

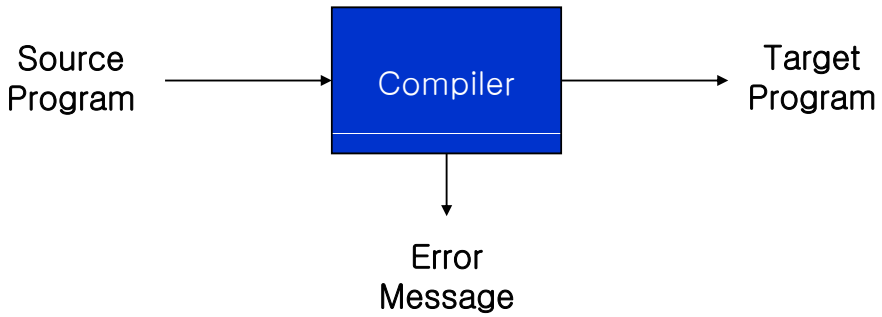
Topics

- Code generation and back-end compilation.
- Memory-oriented software optimizations.

Embedded vs. general-purpose compilers

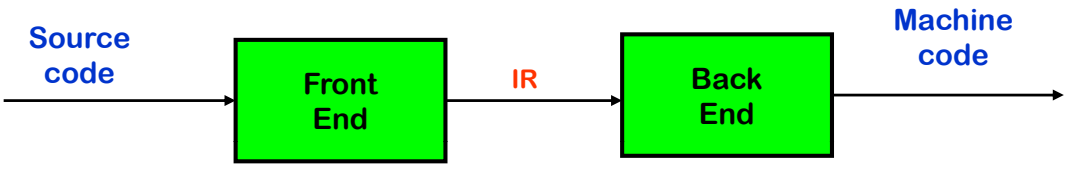
- General-purpose compilers must generate code for a wide range of programs:
  - No real-time requirements.
  - Often no explicit low-power requirements.
  - Generally want fast compilation times.
- Embedded compilers must meet real-time, low-power requirements.
  - May be willing to wait longer for compilation results.

What is a compiler?



- What is an interpreter?
  - A program that reads an *executable* program and produces the results of executing that program

Traditional Two-pass Compiler

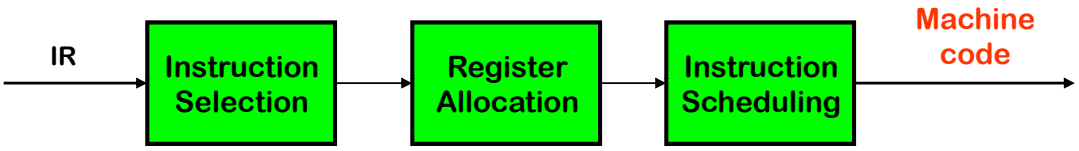


- Use an intermediate representation (IR)
- Front end maps source code into IR
- Back end maps IR into target machine code

Intermediate code generation

- It represents a program for an abstract machine.
- An intermediate representation of the source code program is generated after semantic analysis.
- Three-address code representation is common, in which all memory locations are treated as registers.

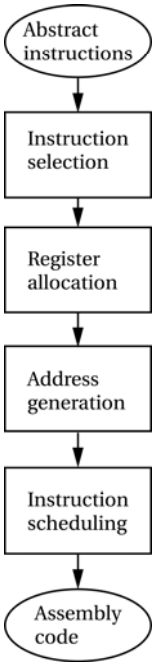
The Back End



- Translate IR into target machine code
- Select instructions to implement each IR operation
- Decide which values are kept in registers
- Automation is less successful in the back end

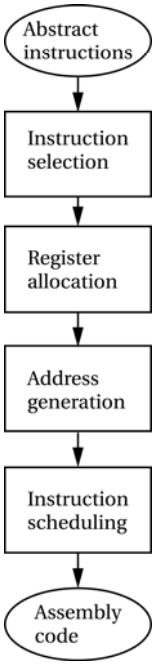
Code generation steps

- Instruction selection chooses opcodes, modes.
  - To minimize code size and execution time
  - A pattern matching problem
- Register allocation binds values to registers.
  - Performed strictly after instruction selection in a general-register machine.
  - Many DSPs and ASIPs have irregular register sets.
  - Optimal allocation is NP-complete



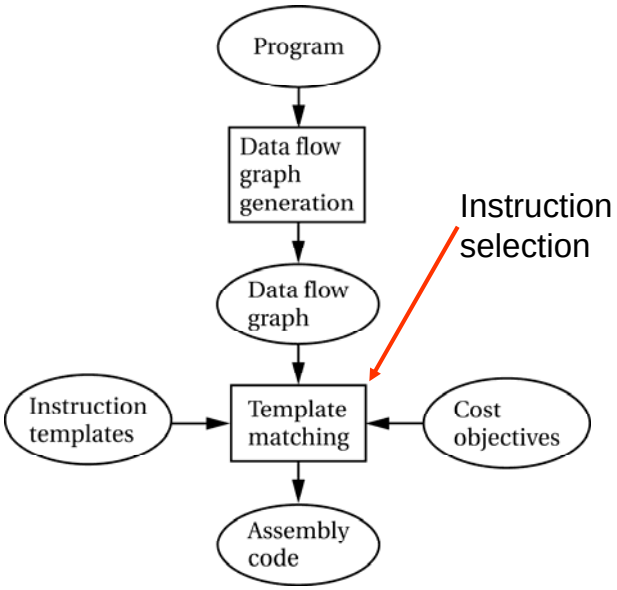
Code generation steps

- Address generation selects addressing mode, registers, etc.
  - Post- or pre-increment addressing for stacks
- Instruction scheduling is important for pipelining and parallelism.
  - Avoid hardware stalls and interlock
  - Use all functional units maximally
  - Optimal scheduling is NP-complete
  - Filling branch delay slots
  - VLIW instruction packet



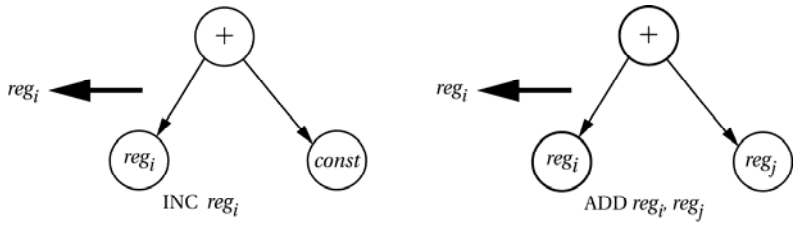
twig model for instruction selection

- twig models instructions, programs as graphs.
- Covers program graph with template matching of instruction graph.
  - Covering can be driven by costs.
  - Use of annotations, such as execution time or energy consumption



twig instruction models

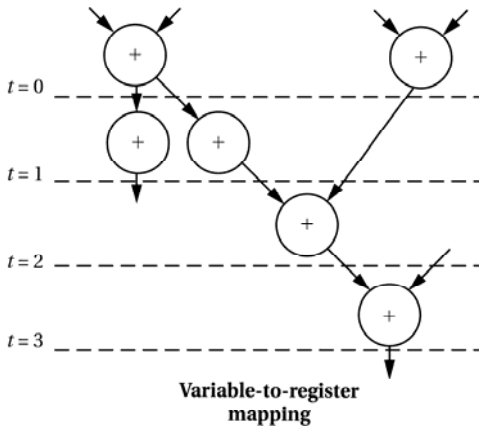
- Rewriting rule:
  - Replacement <- template {cost} = action
- Dynamic programming can be used to cover program with instructions for tree-structured instructions.
  - Must use heuristics for more general instructions.



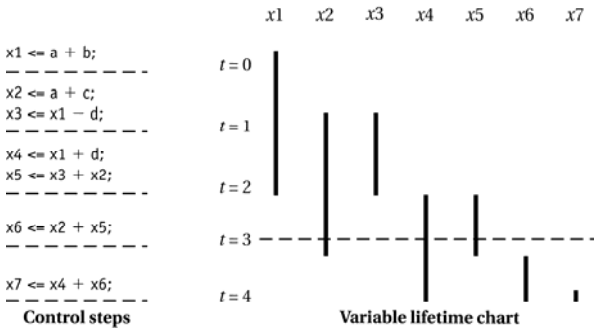
Register lifetimes

- Variable liveness
- V1 and v3 can be assigned to the same registers; similarly, v2 and v4 can be assigned to the same register.
- They are never live at the same time.

```
v1 <= i1;
v2 <= v1 + 1;
v3 <= v2 - i2;
v4 <= v3 - 1;
```
Code

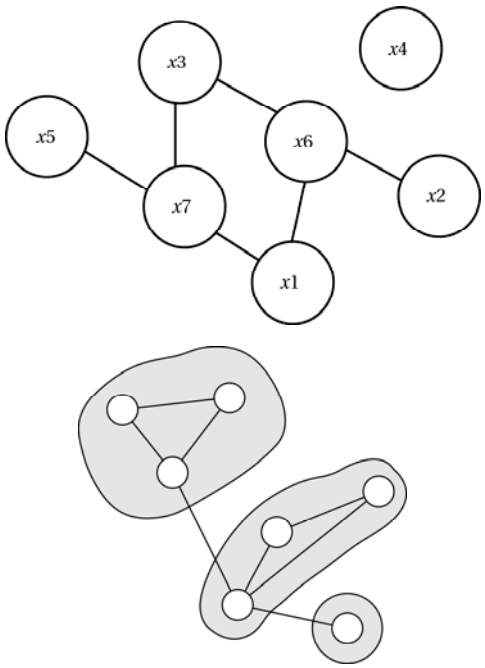


Register lifetimes



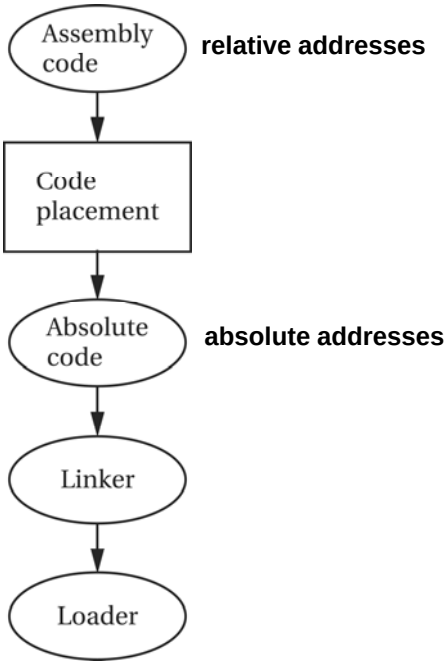
Conflict graph and Clique covering

- Cliques in graph describe registers.
  - Clique: every pair of vertices is connected by an edge.
- Cliques should be maximal.
- Clique covering performed by graph coloring heuristics.



Code placement

- Place code to minimize cache conflicts.
  - Two memory blocks can be mapped into a cache line.
- Possible cache conflicts may be determined using absolute addresses
- May require blank (unused) areas in program.



Linker and Loader

- The linker put together several independently compiled parts into a complete program.
- The loader takes relocatable machine code and alter the addresses, putting the instructions and data in a particular location in memory.

Hwu and Chang

- Analyzed traces to find relative execution times.
- Inline expanded frequently used subroutines to eliminate function call overhead.
- Placed frequently-used traces in the program image by using greedy algorithm.

McFarling procedure inlining

- Estimated number of cache misses in a loop:
  - s_l = effective loop body size.
  - s_b = basic block size.
  - f = average instruction execution frequency of block.
  - M_l = number of misses per loop instance.
  - l = average number of loop iterations.
  - S = cache size.
- Estimated new cache miss rate for inlining;
- Used greedy algorithm to select functions to inline.

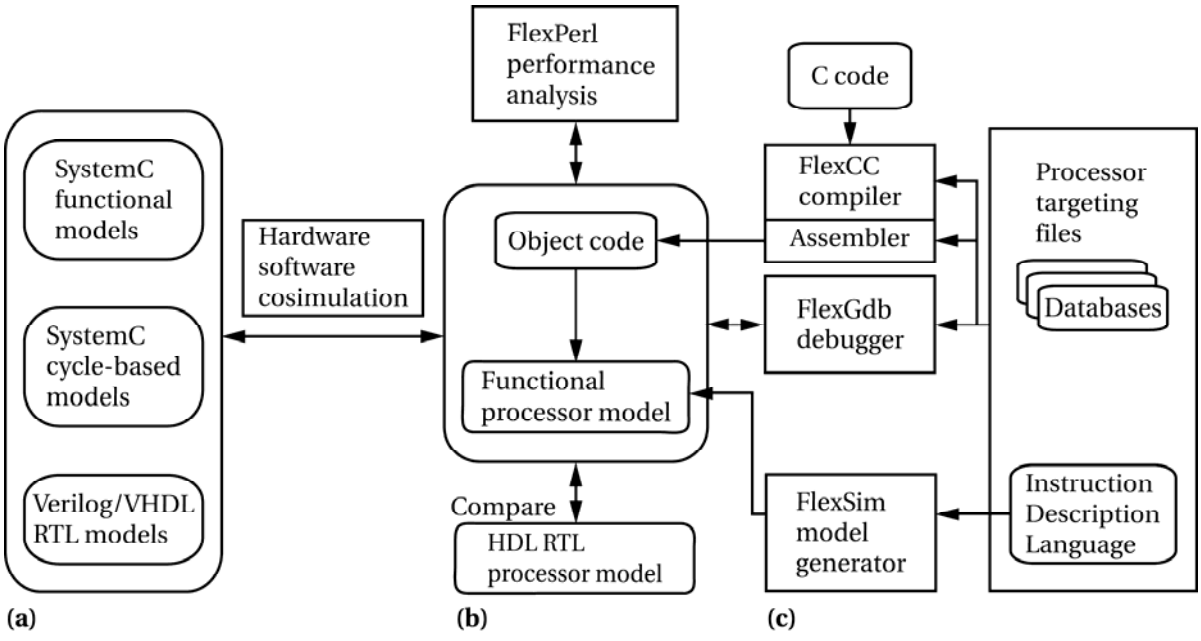
$$s_l = \sum s_b \min(1, f),$$

$$M_l = \max(0, l-1)\max(0, s_l - S),$$

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.
- Rearranged 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.

FlexWare ASIP programming environment



[Pau02] © 2002 IEEE

Memory-oriented optimizations

- Memory is a key bottleneck in many embedded systems.
  - Performance
  - Energy
- Memory usage can be optimized at any level of the memory hierarchy.
  - Various techniques have been developed
  - Recently, optimization for scratch pad memory is developed
- Optimization can target data or instructions.
- Global flow analysis can be particularly useful.
  - Most of embedded systems are composed of many subsystems.
  - Buffers between the subsystems must be carefully sized.

Loop transformations

- Some optimization are applied early during compilation without detailed knowledge of the target hardware.
  - Try to expose parallelism that can be used by later stages.
- Loop-carried dependency

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

Fully parallelizable

```
for (i=1; i<N; i++)
    c[i] = a[i] + c[i−1];
```

Loop-carried dependencies

Loop transformations

- A loop nest has loops enclosed by other loops.
- A perfect loop nest has no conditional statements.
- An imperfect loop nest has conditional that cause some statements in the nest to not be executed in some cases.

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        for (k=0; k<L; k++)
            c[k] = a[i][j] * b[k];
```

Perfect loop nest

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        if (i != j)
            for (k=0; k<L; k++)
                c[k]= a[i][j] * b[k];
```

Imperfect loop nest

Types of loop transformations

- Loop permutation changes order of loops.
- Index rewriting changes the form of the loop indexes.
- Loop unrolling copies the loop body.
- Loop splitting creates separate loops for operations in the loop body.
- Loop merging combines loop bodies.
- Loop tiling splits a loop into a nest of loops, with each inner loop working on a small block of data
- Loop padding adds data elements to change cache characteristics.

Polytope model

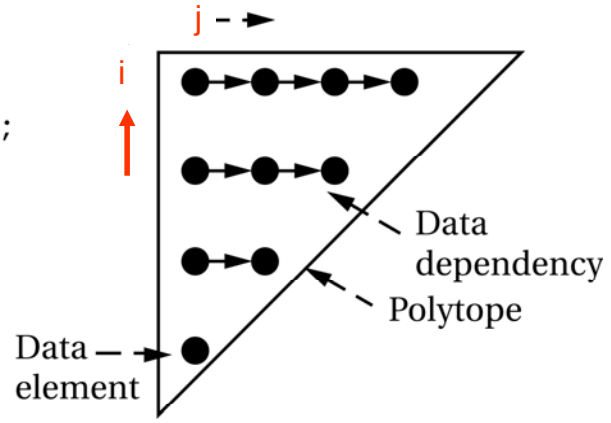
- Commonly used to represent and manipulate the data dependencies in loop nests.
- Loop transformations can be modeled as matrix operations:

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

Loop nest

$$\begin{bmatrix} 0 & 0 \\ N & i \end{bmatrix}$$

Loop nest matrix



Loop permutation and fusion

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

Original loop nest

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

After loop permutation

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

Original loops

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

After loop fusion

Loop fusion

```
for i=0, N
    for j=0,M
        A[i,j]=C[i−2,j+1];
    endfor
    for j=0, M
        B[i,j]=A[i,j+1]+A[i,j]+C[i−1,j];
    endfor
    for j=0, M
        C[i,j]=B[i,j]+A[i,j+2]+A[i,j+1];
    endfor
endfor
```

(a) The original loop with fusion-prevention dependencies.

```
for i=0, N
    A[i,0]=C[i−2,1];
    A[i,1]=C[i−2,2];
    B[i,0]=A[i,0]+A[i,1]+C[i−1,0];
    for j=0, M-2
        A[i,j+2]=C[i−2,j+3];
        B[i,j+1]=A[i,j+2]+A[i,j+1]+C[i−1,j+1];
        C[i,j]=B[i,j]+A[i,j+1]+A[i,j+2];
    endfor
    B[i,M]=A[i,M]+A[i,M+1]+C[i−1,M];
    C[i,M−1]=B[i,M−1]+A[i,M]+A[i,M+1];
    C[i,M]=B[i,M]+A[i,M+1]+A[i,M+2];
endfor
```

(b) The fused loop after transformation.

Kandemir et al. loop energy experiments

ABSTRACT

High-level compiler optimizations have been widely used to achieve speedups on array-based codes. Such optimizations are becoming increasingly important in embedded signal processing and multimedia systems. The focus of these optimizations has traditionally been on improving performance. However, energy constraints are of critical importance in battery-operated embedded devices. In this paper, we present an experimental evaluation of several state-of-the-art compiler optimizations on energy consumption, considering both the processor core (datapath) and memory system. This is in contrast to many of the previous works that have considered them in isolation.

Program	Array Sizes	Miss Rate	Optimizations
adi	100*100*2	0.0979	linear transforms, tiling
hydro2d/fct	52*52	0.0962	loop fusion
nasa7/btrix	100*100*100*5	0.2063	loop fusion
nasa7/cholesky	52*52	0.1109	loop fission
tomcatv	100*100	0.2403	scalar expansion

Table 1: Programs used in the experiments.

Kandemir et al. loop energy experiments

A C source benchmark is compiled by the *SimpleScalar* version of `gcc`, which generates *SimpleScalar* assembly codes. The *SimpleScalar* assembler `gas` and loader/linker `gld` produce *SimplePower* executables that can then be loaded into *SimplePower* main memory and executed by *SimplePower core*. In our study, we enhanced a source-to-source optimizer [4] to perform the various code transformation investigated. The simulator can be configured using the command line to set the caches parameters, output the pipeline trace cycle-by-cycle, and dump the memory image. *SimplePower* provides the register file final status, total number of cycles in execution, number of transitions in on-chip buses, switch capacitance statistics for each pipeline stage, switch capacitance statistics for different functional units, and the total switch capacitance.

Kandemir et al. loop energy experiments

In `adi`, the linear loop optimizations interchanged the order of two loops in the main nest in an attempt to obtain stride-one accesses in the innermost loop, thereby improving spatial locality. (The original version is denoted by `orig` and the tiled version is denoted by `tile`). When we enable tiling, the compiler tiled the innermost loop and hoisted the tile loop (i.e., the loop that iterates on tiles) to the outermost position. Note that in order to run this code standalone, we added a two-deep initialization nest (i.e., a nest that contains two nested loops). The linear transformation permuted this nest whereas tiling did not modify it due to its relatively small contribution to the overall performance.

Kandemir et al. loop energy experiments

	Core Energy (J)	Memory Energy (J)			
		↓→	1-way	2-way	4-way	8-way
orig	0.0043	1K	0.1604	0.0915	0.0794	0.0772
		2K	0.1159	0.0789	0.0756	0.0757
		4K	0.1000	0.0763	0.0759	0.0760
		8K	0.0730	0.0681	0.0742	0.0766
loop	0.0054	1K	0.1418	0.0630	0.0526	0.0468
		2K	0.0844	0.0493	0.0435	0.0436
		4K	0.0609	0.0441	0.0439	0.0440
		8K	0.0378	0.0283	0.0231	0.0251
tile	0.0052	1K	0.1404	0.0731	0.0729	0.0688
		2K	0.0942	0.0646	0.0674	0.0689
		4K	0.0550	0.0426	0.0465	0.0457
		8K	0.0345	0.0228	0.0220	0.0221

Table 2: Energy consumption in `adi`.

Kandemir et al. loop energy experiments

In the `adi` code, the number of memory accesses per computation is very high. This is because the it accesses three-dimensional arrays using two-deep loop nests. Consequently, as shown in Table 2, the core power is very low compared to memory power for all cache configurations. An optimizing compiler can be very aggressive in applying potential optimizations such as tiling, if it can detect that the number of memory references per computation is very high. However, we note that applying tiling increases the core energy consumption.

Kandemir et al. loop energy experiments

In `nasa7/btrix`, in order to isolate the impact of loop fusion, we disabled other loop optimizations, and experimented with only original (`orig`) and fused (`fuss`) versions. When fusion is activated, the compiler fused two large one-dimensional (one-deep) loop nests into a very large loop. This example gives us the opportunity for observing the impact of loop fusion (in its extreme, when the resulting loop body gets very large and the chances for intra- and inter-array conflict misses in the data cache increase greatly) on power dissipation. In `hydro2d/fct`, we again measured the impact of fusion on power consumption using the original (`orig`) and the fused (`fuss`) versions. This time the compiler fused both initialization nests (three of them) as well as two main loop nests (each two-deep). In comparison to `nasa7/btrix`, the resulting loop bodies are not very large.

Kandemir et al. loop energy experiments

	Core Energy (J)	Memory Energy (J)				
		↓	1-way	2-way	4-way	8-way
orig	0.1565	1K	8.4840	3.9372	2.8179	2.9734
		2K	3.3221	2.3311	1.3897	1.2614
		4K	1.6816	1.3939	1.1123	1.1155
		8K	1.3291	0.9573	0.8942	0.8752
fuss	0.1748	1K	9.4086	4.4788	3.1901	3.2580
		2K	3.9100	2.5048	1.4469	1.2732
		4K	1.8087	1.4712	1.1003	1.1033
		8K	1.3887	0.9480	0.8852	0.8652

Table 3: Energy consumption in `nasa7/btrix`.

Kandemir et al. loop energy experiments

	Core Energy (J)	Memory Energy (J)			
		$\downarrow^{\rightarrow}$	1-way	2-way	4-way	8-way
orig	0.0008	1K	0.0290	0.0117	0.0079	0.0079
		2K	0.0130	0.0069	0.0060	0.0054
		4K	0.0086	0.0055	0.0054	0.0054
		8K	0.0066	0.0055	0.0055	0.0053
fuss	0.0006	1K	0.0277	0.0102	0.0073	0.0068
		2K	0.0095	0.0074	0.0061	0.0063
		4K	0.0069	0.0050	0.0050	0.0050
		8K	0.0057	0.0050	0.0050	0.0050

Table 4: Energy consumption in `hydro2d/fct`.

Kandemir et al. loop energy experiments

In the next two examples, we evaluated the impact of loop fusion on core and memory system energy consumption. In `nasa7/btrix`, the loop fusion interferes with loop scheduling as the loop body becomes very large. This increases the core power (by 12%) as well as memory power (due to poor scheduling of memory operations) as shown in Table 3. Unfortunately, scalar replacement could not eliminate the large number of memory references. On the other hand, if the cache size is large or the associativity is high, scheduling memory operations becomes less critical (unless the cache is direct-mapped, in which case poor scheduling induces more conflict misses). For example, for a 4K, 4-way set associative cache, the fused version is marginally better than the original. In the `hydro2d/fct` case, the same optimization is more successful from the energy point of view (see Table 4). It reduces the core power as well as the memory system power by as much as 25%. The reduction in the memory power comes from reductions in the number of memory references rather than from hit/miss rate variations. Overall, we believe that, if applied judiciously, the loop fusion can reduce both the core and memory system power.

Catthoor et al. methodology

- It is for streaming systems such as multi-media
  - Memory-oriented data flow analysis and model extraction, which analyzes loops to identify memory requirements.
  - Global data flow transformations to improve memory utilization.
  - Global loop and control flow optimizations to eliminate system-level buffers and improve data locality
  - Data reuse decisions for memory hierarchy exploits caches to reduce energy consumption and improve performance
  - Memory organization designs the memory systems and its ports
  - In-place optimization use low-level techniques to reduce storage requirements

Buffers

- Buffers mediate between subsystems
- Producers: subsystems that generate data
- Consumers: subsystems that consume data
- Buffers make sure that all data are delivered form the producer to the consumer
- The buffers must be sized properly

Buffer management

- Excessive dynamic memory management wastes cycles, energy with no functional improvements.
- IMEC: analyze code to understand data transfer requirements, balance concerns across program.
- Panda et al.: loop transformations can improve buffer utilization.
  - Reuse b more easily
  - Easier for pre-fetching

- Before:

```
for (i=0; i<N; ++i)
    for (j=0; j<N-L; ++j)
        b[i][j] = 0;
for (i=0; i<N; ++i)
    for (j=0; j<N-L; ++j)
        for (k=0; k<L; ++k)
                b[i][j] = a[i][j+k];
```

- After:

```
for (i=0; i<N; ++i)
    for (j=0; j<N-L; ++j)
        b[i][j] = 0;
        for (k=0; k<L; ++k)
                b[i][j] = a[i][j+k];
```

closer

Cache optimizations

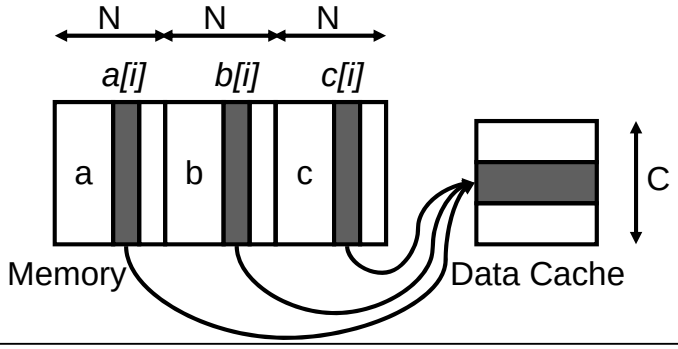
- Strategies:
  - (Improve hit rate) rearrange data to reduce the number of conflicts.
  - rearrange data to take advantage of prefetching.
- Need:
  - Load map.
  - Information on access frequencies.

Cache data placement

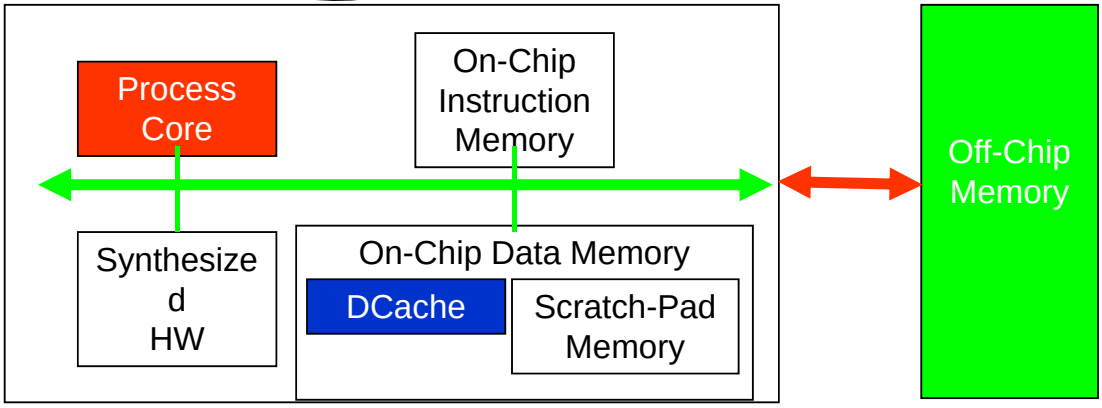
- Panda & Dutt: rearrange data to reduce cache conflicts. It is for scalar variables.

1. Build a closeness graph for accesses by using the access patterns of the variables.
2. Cluster variables into cache-line sized units.
3. Build a cluster interference graph.
4. Use interference graph to optimize placement.

Problem Description




```
int a[N], b[N], c[N]
...
for i = 0 to N-1
      c[i] = a[i] + b[i]
end for
```

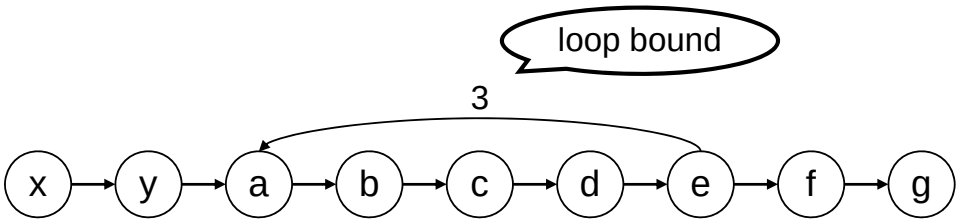


Memory Data Organization of Scalar Variables

- Assumption
  - scheduling and register allocation already performed
  - sequence of accesses to variables is fixed
- Steps
  - Build Closeness Graph
  - Group the variables into clusters of *L* words (*L*: cache line size)
  - Build a Cluster Interference Graph (*CIG*)
  - Assign memory locations to clusters

Memory Organization of Scalar Variables

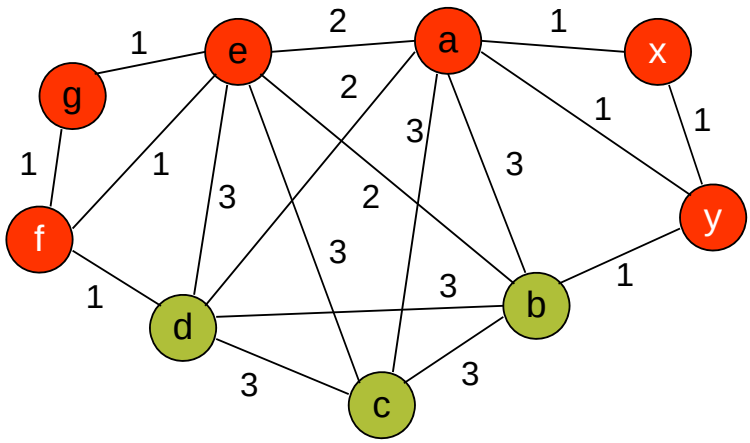
- Generate an Access Sequence



distance(u, v) = number of distinct variable nodes encountered on a path from u to v, or v to u (including u and v)

Memory Organization of Scalar Variables

- Construct a Closeness Graph of the variables



Closeness Graph for loop bound = 3

Grouping of Variables into Clusters

Procedure *PeformClustering*
Input: *CG(V, E)* - Closeness Graph; *L* - Cache Line Size
Output: Set *F* - Set of clusters of size *L*

```
for each vertex u in V
    Find the sum of incident edge weights S(u) = Σ_{v∈V} e(u,v)
end for
Let X = vertex set V.
while (X ≠ φ) do
    Let u = vertex v ∈ X with maximum S(v)
    Create new cluster C = {u}
    while (size of cluster C≠ L) and (X ≠ φ) do
        Let x be the variable ∈ X with maximum value for T,
        where T = Σ_{u∈C, v∈X-C} e(u,v)  -- i.e., x is the variable with maximum sum
        C = C ∪ {x};   X = X - {x}        -- of edge weights with nodes already in C
    end while
    Set e(u, v) = 0   ∀(u∈C) or (v∈C)
      -- i.e., delete all edges connecting to vertices in cluster C just formed
    Update S(v)  ∀(v∈X)
end while
end Procedure
```

Cluster Interference Graph (CIG)

- Procedure for generating the CIG

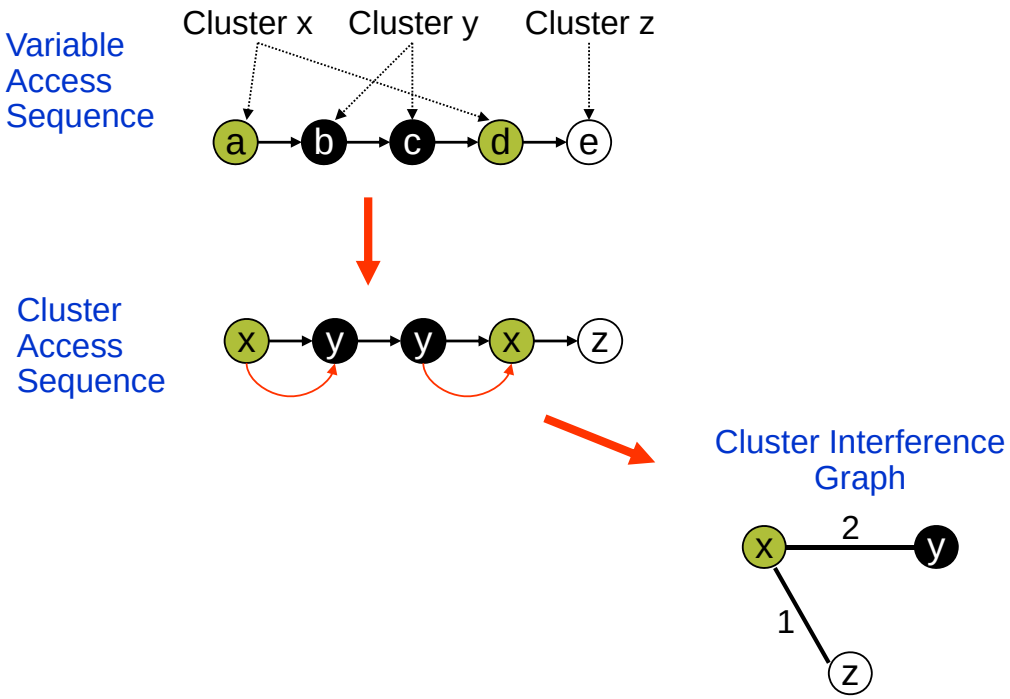
Procedure *BuildCIG*

Input: *A* - Variable Access Sequence; *F* - Set of Clusters
Output: *CIG* - Cluster Interference Graph

Convert the Variable Access Sequence *A* into a Cluster Access Sequence by renaming each node *u* in the sequence by the cluster *C*, where $u \in C$.
Create a node in *CIG* for each cluster in *F*.
Assign edge weight *e(u,v)* between nodes *u* and *v* = the number of times the access to cluster *u* and *v* alternate along the execution path.

end Procedure

Cluster Interference Graph (CIG)



Memory Location Assignment

- The cost of a *memory assignment* :

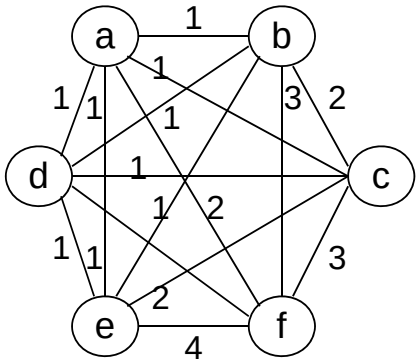
$$MemAssignCost(CIG) = \sum_{x,y \in V(CIG)} e(x,y) \times P(x,y)$$

where *e(x,y)* is the edge weight, and

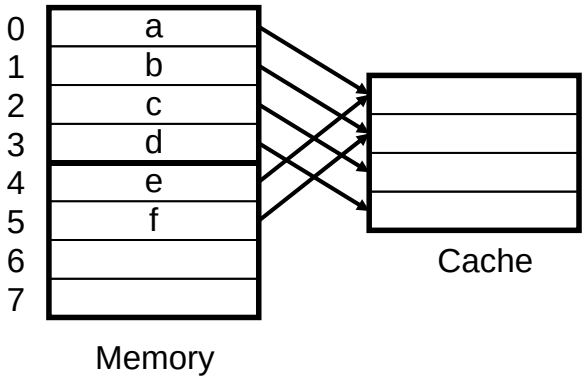
P(x,y) = 1 if memory locations for x and y map into the same cache line
0 otherwise

- In order to minimize conflict misses, we need to solve the following Cluster Assignment problem:
  - Find an assignment of clusters in a CIG to memory locations, such that MemAssignCost (CIG) is minimized.

Memory Location Assignment



(a) CIG



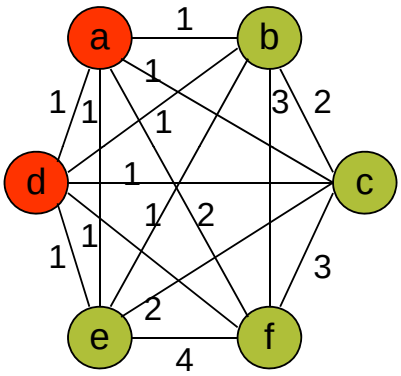
(b) Memory Assignment

Conflicting Pairs	Cost
a, e	1
b, f	3

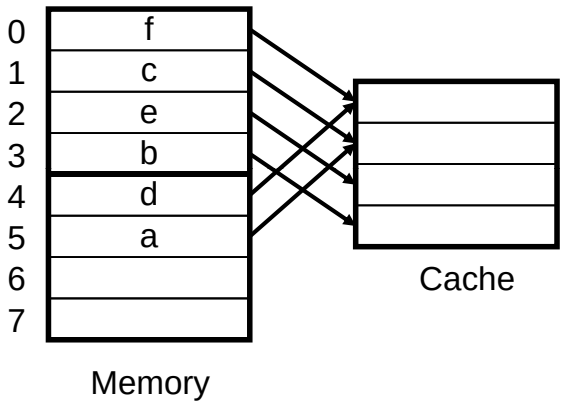
Total Cost = 1 + 3 = 4

(c) Cost Assignment

Memory Location Assignment



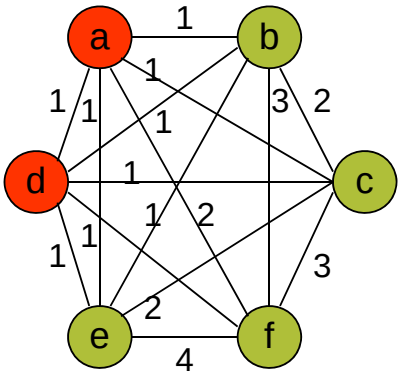
(a) CIG



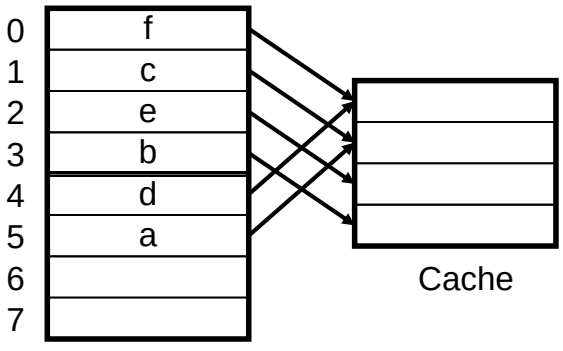
(b) Memory Assignment

S(f) = 15, S(c) = 9, S(e) = 9, S(b) = 8 ,S(d) = 5, S(a) =5

Memory Location Assignment



(a) CIG



(b) Memory Assignment

Conflicting Pairs	Cost
f, d	1
c, a	1

Total Cost = 1 + 1 = 2

(c) Cost Assignment

cost(a,0) = 2 cost(d,0) = 1
cost(a,1) = 1 cost(d,1) = 1
cost(a,2) = 1 cost(d,2) = 1
cost(a,3) = 1 cost(d,3) = 1

Memory Location Assignment

- Procedure for assigning clusters to memory locations

Procedure *AssignClusters*
Input: *CIG(V,E)* - Cluster Interference Graph
Output: Assignment of Clusters to Memory Locations

Sort the vertices of *CIG* in descending order of *S(u)*
-- *S(u) is the sum of edge weights incident on vertex u*
Let *X* be this sorted list of vertices
while ($X \neq \phi$) **do**
Create new page *P* in memory
while (size of page *P* < *k*) and ($X \neq \phi$) **do**
u = head of list *X*
Assign *u* to line *i* of page *P*, where *cost(u, i)* is minimum over *i=0 ... k-1*
Delete *u* from *X*
end while
end while
end Procedure

Memory Organization for Array Variables

- Objective
  - Minimizing data cache conflict misses
  - The problem of clustering of variables to avoid compulsory misses is not relevant.
- Steps
  - Constructing the Interference Graph
  - Memory Assignment to Array Variables

Constructing the Interference Graph

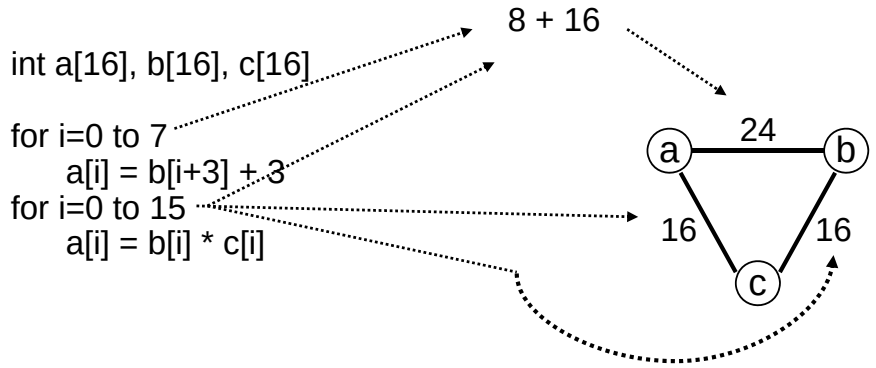
- If two arrays A and B are accessed repeatedly within a loop, then there is a possibility that accesses to A and B might cause conflict misses in the data cache.
- Procedure for building Interference Graph for arrays

Procedure *BuildArrayIG*
Input: Code with array accesses
Output: Interference Graph IG of arrays

Create a node *u* for every array *u* in the code
Initialize edge weights *e(u,v) = 0* for all *u, v*
for all (innermost) loops *l* in the code **do**
Let *L* be the loop bound of loop *l*
Let *X* = set of all arrays accessed in *l*
Update *e(u, v) = e(u, v) + L* for all $u,v \in X$
end for
end Procedure

Memory Assignment to Array Variables

- Interference Graph: edge weights contributed by loop bounds

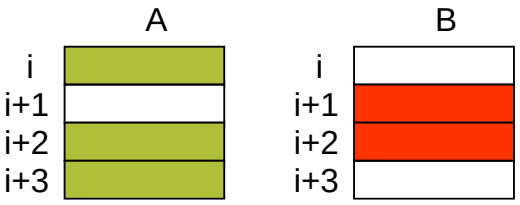


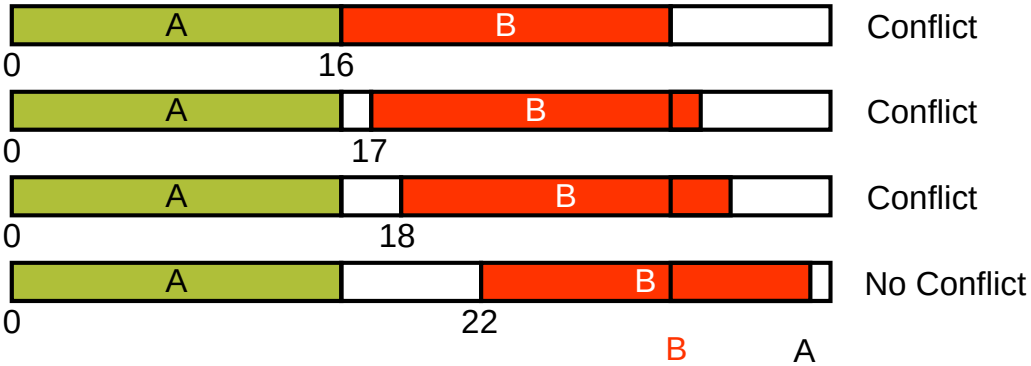
Two memory locations X and Y will map into the same cache line in a direct-mapped cache with k lines (L words per line), if

$$\left(\left\lfloor \frac{X}{L} \right\rfloor - \left\lfloor \frac{Y}{L} \right\rfloor\right) \bmod k = 0 \quad i.e., \quad (nk-1) < \frac{X-Y}{L} < (nk+1) \qquad (4.1)$$

Memory Assignment to Array Variables

```
int A[16], B[16];
for i=0 to 12
    A[i] = A[i+2] + A[i+3]
         + B[i+1] + B[i+2];
```





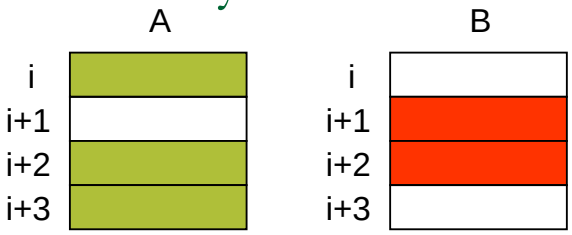
$$(nk-1) < \frac{X-Y}{L} < (nk+1) \qquad (4n-1) < \frac{(22+i+1)-(i+3)}{4} < (4n+1)$$

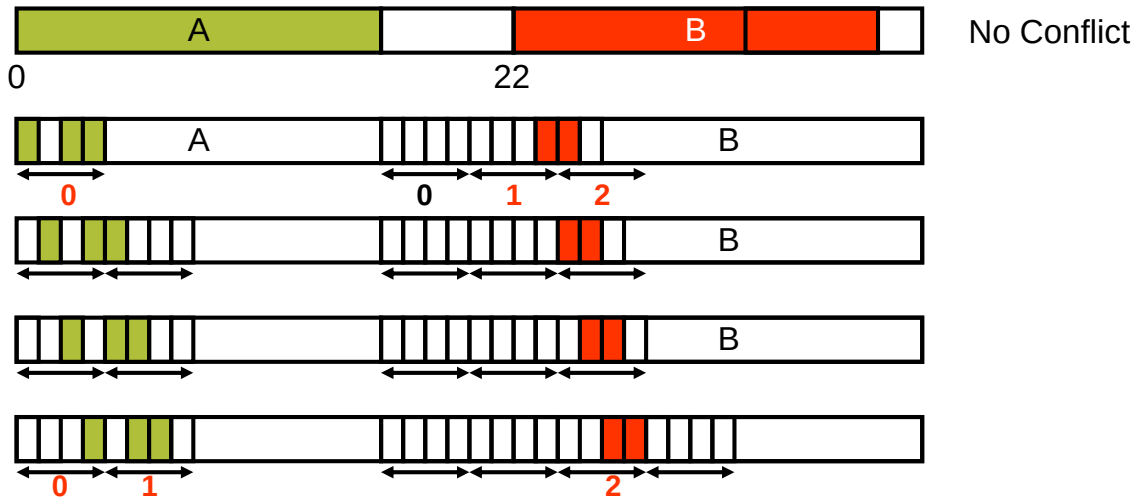
(B: 22 + i + 1; A: i + 3)

k = 4, L = 4 Worst case

Memory Assignment to Array Variables

```
int A[16], B[16];
for i=0 to 12
    A[i] = A[i+2] + A[i+3]
         + B[i+1] + B[i+2];
```





Memory Assignment to Array Variables

- Cost function for expected number of conflicts

Function *AssignmentCost*
Input: *u* - array under test; *A* - proposed start address; Access Sequence; Array assignment already completed; *IG* - Interference Graph
Returns: Expected number of cache conflicts for this assignment

```
Initialize cost = 0
for all v_l | e(v_l ,u) ≠ 0, v_l already assigned
  -- i.e., all assigned arrays that have an edge with u in IG
    for each loop (bound L) in which accesses to v_l and u occur
        w = no. of times control alternates between elements of v_l and u
        that map into the same cache line, using Condition (4.1)
        cost = cost + w * L       -- w = 0 if there is no conflict
    end for
  end for
  return cost
end Function
```


Memory Assignment to Array Variables

- Procedure for assigning addresses to arrays

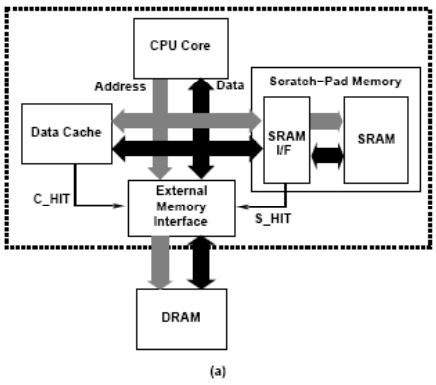
Procedure *AssignArrayAddresses*
Input: *IG* - Interference Graph; *k* - number of cache lines
Output: Assignment of addresses to all arrays (nodes in *IG*)

```
Address A = 0
Sort nodes in IG in decreasing order of S(u) (sum of incident edge weights)
Let the list of nodes be: v_0 ... v_n-1
for i = 0 ... n-1
    Initialize cost c = ∞
    min = 0          -- keeps track of cache line with minimum mapping cost
    for j = 0 ... k-1
        if AssignmentCost(v_i , A+j) < c then
            c = AssignmentCost(v_i , A+j)
            min = j
        end if
    end for
    Assign address (A+min) to first element of v_i
    A = A + min + arraysize(v_i)          -- updating A for next iteration
  end for
end Procedure
```

Memory Assignment to Array Variables

- This cost is equal to the expected number of cache conflicts with all arrays that have already been assigned.
- If the conflict condition does not resolve to a constant, then we conclude that the two arrays do not conflict.

Scratch Pad Memory



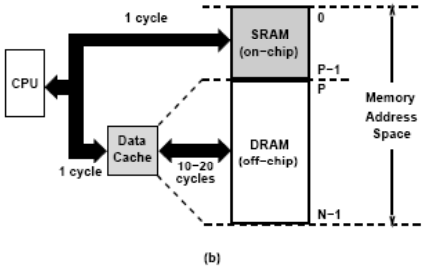


Figure 1. (a) Block Diagram of Embedded Processor Configuration (b) Division of Data Address Space between SRAM and DRAM

Scratch Pad Memory

- The accesses to Hist are data-dependent

```
char BrightnessLevel [512][512];
int Hist [256];  /* Elements initialized to 0 */
. . .
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        /* For each pixel (i, j) in image */
        level = BrightnessLevel [i][j];
        Hist [level] = Hist [level] + 1;
```