Embedded Systems Design with Platform FPGAs

Chapter -4 Partitioning

Dr. Bassam Jamil
Adopted and updated from
Ron Sass and Andrew G. Schmidt

Chapter 4 — Partitioning

Outline of This Chapter

Chapter 4 Learning Objectives

Topics

- partitioning hardware and software tasks;
- formulation of an analytical solution;
- pragmatic issues

Problem Statement

Given a Software Reference Design, how to decompose it into hardware and software components?

Key Elements:

- software reference design is often a sequential application
- goal in an embedded system is to address one (or more) of the multi-faceted performance metrics (from chapter 1)
- FPGA’s resources fungible but there is fixed total

Our Approach

- formulate the problem analytically
- explore mathematical optimization of model
- describe practical issues not accounted for in model

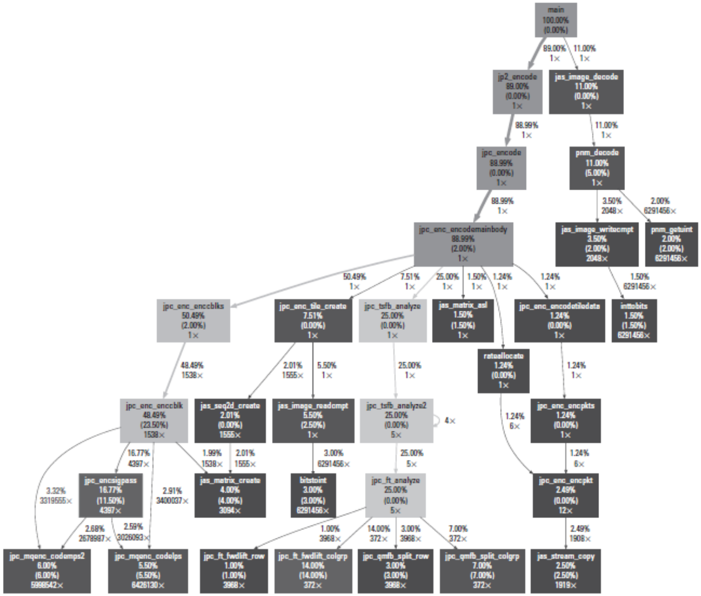
4.1 Overview of Partitioning Problem

- Partitioning is:
  - the **grouping** of specific sets of instructions in an application
  - and then **mapping** those groups to either hardware or software.
- The result of partitioning is known as decomposition.
- Factors to guide partitioning decisions:
  - Expected **performance gains** (due to moving software into hardware),
  - **Resources** used in the hardware implementation,
  - how often that portion of the application is used
  - (most importantly) how much **communication overhead** the decomposition introduces.

Design Feature

- A feature is a connected **cluster of instructions** from the application's software reference design suitable for a hardware implementation.
- Think of a feature as a **subroutine** from the software reference design.
- To achieve good partitioning, we generally have to examine groupings that may be bigger than or smaller than the subroutines defined by the programmer.
- Because feature size affects performance, the decision of whether to **implement a feature in hardware depends on its improvement to the overall system** performance and the resources it uses relative to the other candidate features.

Design Example: JPEG2000 Encoding


main 100.00% (0.00%)
89.00% 1×
11.00% 1×
jp2_encode 89.00% (0.00%) 1×
jas_image_decode 11.00% (0.00%) 1×
88.99% 1×
11.00% 1×
jpc_encode 88.99% (0.00%) 1×
pnm_decode 11.00% (5.00%) 1×
88.99% 1×
3.50% 2048×
2.00% 6291456×
jpc_enc_encodemainbody 88.99% (2.00%) 1×
jas_image_writecmpt 3.50% (2.00%) 2048×
pnm_getuint 2.00% (2.00%) 6291456×
50.49% 1×
7.51% 1×
25.00% 1×
1.50% 1×
1.24% 1×
1.24% 1×
1.50% 6291456×
jpc_enc_encodeblks 50.49% (2.00%) 1×
jpc_enc_tile_create 7.51% (0.00%) 1×
jpc_tsfb_analyze 25.00% (0.00%) 1×
jas_matrix_asl 1.50% (1.50%) 1×
jpc_enc_encodetiledata 1.24% (0.00%) 1×
inttobits 1.50% (1.50%) 6291456×
ratealloate 1.24% (0.00%) 1×
48.49% 1538×
2.01% 1555×
5.50% 1×
25.00% 1×
1.24% 1×
jpc_enc_encodeblk 48.49% (23.50%) 1538×
jas_seq2d_create 2.01% (0.00%) 1555×
jas_image_readcmpt 5.50% (2.50%) 1×
jpc_tsfb_analyze2 25.00% (0.00%) 5×
4×
1.24% 6×
jpc_enc_encpkts 1.24% (0.00%) 1×
16.77% 4397×
1.99% 1538×
2.01% 1555×
3.00% 6291456×
25.00% 5×
1.24% 6×
3.32% 3319555×
jpc_encsigpass 16.77% (11.50%) 4397×
2.91% 3400037×
jas_matrix_create 4.00% (4.00%) 3094×
bitstoint 3.00% (3.00%) 6291456×
jpc_ft_analyze 25.00% (0.00%) 5×
jpc_enc_encpkt 2.49% (0.00%) 12×
2.68% 2678987×
2.59% 3026093×
1.00% 3968×
14.00% 372×
3.00% 3968×
7.00% 372×
2.49% 1908×
jpc_mqenc_codemps2 6.00% (6.00%) 5998542×
jpc_mqenc_codelps 5.50% (5.50%) 6426130×
jpc_ft_fwdlift_row 1.00% (1.00%) 3968×
jpc_ft_fwdlift_colgrp 14.00% (14.00%) 372×
jpc_qmfb_split_row 3.00% (3.00%) 3968×
jpc_qmfb_split_colgrp 7.00% (7.00%) 372×
jas_stream_copy 2.50% (2.50%) 1919×

Design Example

- The graph illustrated in Figure shows the execution of a JPEG2000 encoder application.
  - From the main subroutine, we can see that 11% of the time is spent reading the source image and 89% of the time is doing the encoding.
  - Encoding is not a single subroutine but rather a score of subroutines.
- If communication were free and hardware resources limitless, one would simply move everything into hardware.
- Given that neither is true, we have to **determine which subroutines (or parts of subroutines) will have the biggest impact on some performance metric**.

Profiling

- To profile an application means to **collect run-time information** on an application during execution.
- With profiling:
  - the software reference design is executed with representative input and
  - the time spent in various parts of the application is measured.
- Profiling can be accomplished in a number of ways.
- We assume the **simplest technique**:
  - that profiles an application by **periodically interrupting** the program and sampling the program counter.
  - A **histogram** is used to count how often a program is interrupted at a particular address in the application.
  - From histogram data, an **approximate fraction** of total execution time spent in various parts of the application can be computed.
- The Unix tool for profile: **gprof**

Performance Analysis

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

- Amdahl’s law applies the law of diminishing returns to the usefulness of a single architecture feature.
  - It articulates the limits of the overall improvement to an application that a single enhancement can make on the execution time.
- Can we generalizing the to include all potential enhancements?
- Answer: Amdahl’s law is **not suitable** because:
  - It **addresses a single enhancement** and doesn’t help us select a subset from a collection of potential features.
  - It focuses solely on execution time
  - It **does not address resources** required.
  - It **does not address communication** costs

Outline of This Chapter

4.2. Analytical Solution to Partitioning

- The goal of the analytical solution:
  - formulates the problem of **grouping** instructions into features and then **mapping** those features to either hardware or software.
  - The objective is to optimize a performance metric. We will try to maximize one metric: speedup.
- The accuracy:
  - mathematical solution will produce an approximate answer to the partitioning problem. Why?
    - First: Not all of these issues can be incorporated into an analytical model
    - Second: many of the inputs to our model are estimates or approximations themselves
- If it is not accurate then why bother?
- We get a partitioning that is close to optimal → Good starting point
  - From there, it is up to the designer to be artful in their use of guidelines and engineering expertise to refine the solution.

4.2.1. Basic Definitions: application , partition

- A subroutine of a software reference design is modeled as a Control Flow Graph (CFG).
- a Call Graph (CG) consists of a set of CFGs (one per subroutine):
  - ***C*** = {C0, C1, . . . , Cn−1}
  - where Ci = (Bi, Fi) is the CFG of subroutine i.
- **The application**'s (static) CG is then written:
  - A = (***C***,L)
  - where L ∈ C ×C.
  - Two subroutines are related (Ci, Cj) ∈ L if it can be determined at compile time that subroutine i has the potential to invoke subroutine j.
- **Partition S = {S0, S1, . . .}** of some universal set U is a set of subsets of U such that:

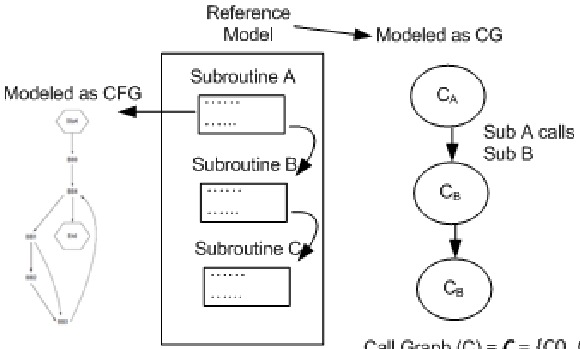
$$\bigcup_{S\in\mathcal{S}} S = U \qquad (4.1)$$

$$\forall S, S' \in \mathcal{S} \mid S \cap S' = \emptyset \qquad (4.2)$$

$$\forall S \in \mathcal{S} \cdot S \neq \emptyset \qquad (4.3)$$

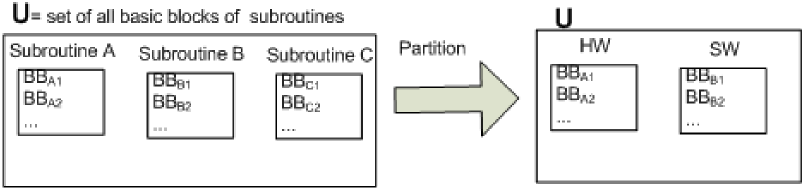
Equation 4.1 says that every element of U is a member of at least one subset S ∈ ***S***.
Equations 4.2 and 4.3 say that the subsets S ∈ ***S*** are pair-wise disjoint and not empty.
In other words, **every element of our universe U ends up in exactly one of the subsets of *S* and none of the subsets are empty**.

4.2.1. Basic Definitions: application , partition



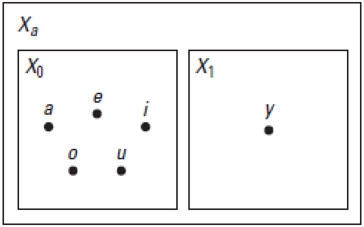
Call Graph (C) = **C** = {C0, C1, . . . , Cn−1}
Application (A)=(C,L)

U= set of all basic blocks of subroutines



Example of Universal Set ***U* and *S***

- Consider the set of English vowels, U = {a, e, i, o, u, y}.
- Examples of **legal partitions**:
  - Xa = {{a, e, i, o, u}, {y}}
  - Xb = {{a}, {e}, {i}, {o}, {u}, {y}}.



$\mathcal{X}_a$ illustrated graphically.

- **Illegal Partitions**:
  - Xc = {{a}, {e}, {i}, {o}}
    - Xc violates Equation 4.1, it does not include “u”
  - Xd = {{a, e, i}, {i, o, u, y}, {}}
    - Xd violates both Equation 4.2 and Equation 4.3. It has empty set and sets have common elements.

Examples of Partition of a Set:

Set	Partition	# partitions
Empty Set : {}	X_0={}	1
Singleton Set: {a}	X_0={ {a} }	1
{ a, b}	X_0= {{a}, {b}} X_1= { {a, b} }	2
{a, b, c}	X_0= { {a}, {b}, {c} } X_1= { {a, b}, {c} } X_2= { {a, c}, {b} } X_3= { {a}, {b, c} } X_4= { {a, b, c} }	5

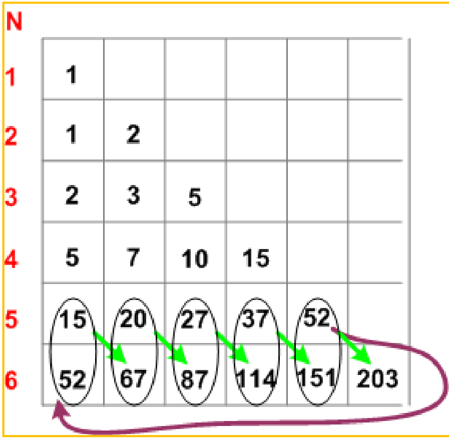
Partition of a Set= {a, b, c, d}

Set	Partition	Partitions
{a, b, c, d}	X_0= { {a}, {b}, {c}, {d} } X_1= { {a, b}, {c}, {d} } X_2= { {a, c}, {b}, {d} } X_3= { {a, d}, {b}, {d} } X_4= { {b, c}, {a}, {d} } X_5= { {b, d}, {a}, {c} } X_6= { {c, d}, {a}, {b} } X_7= { {a, b}, {c, d} } X_8= { {a, c}, {b, d} } X_9= { {a, d}, {b, c} }	X_{10}= { {a, b, c}, {d} } X_{11}= { {a, b, d}, {c} } X_{12}= { {a, c, d}, {b} } X_{13}= { {b, c, d}, {a} } X_{14}= { {a, b, c, d} } **Number of partitions is 15**

Computing Number of Partitions

- Number of partitions are computed using **Bell triangle.**
- **Constructing Bell triangle:**

1. The first row starts with value of “1”
2. Next rows are computed as follows:
   a) First value in each row is copied from the end of the previous row,
   b) Next values are computed by adding the two numbers to the left and above left of each position.
   c) The number of partition is end of the row



N						
1	1					
2	1	2				
3	2	3	5			
4	5	7	10	15		
5	15	20	27	37	52	
6	52	67	87	114	151	203

Application Partition

- Assume our universe is the set of all of the basic blocks from every subroutine

$$U = \bigcup_{C \in \mathcal{C}} V(C)$$

- then the subroutines is a partition in *U*

$$\mathcal{S} = \left\{ \underbrace{\{b_0, b_1, \ldots, b_i\}}_{\text{subroutine } C_0}, \underbrace{\{b_i, b_{i+1}, \ldots, \}}_{\text{subroutine } C_1}, \cdots \underbrace{\{b_j, b_{j+1}, \ldots, \}}_{\text{subroutine } C_{n-1}} \right\}$$

- Our task in partitioning is:
    - To reorganize the partition of basic blocks . We can create new (nonempty) subsets, remove subsets, and move basic blocks around until we have a new partition . The result is **a new application A' = (C' ,L')** inferred from the reorganized partition X'.
    - The second step is to **map** each subset of X to either hardware or software:

$$\mathcal{X}' = \left\{ \underbrace{\{b_j, b_{j+1}, \ldots, \}\{b_k, b_{k+1}, \ldots, \} \cdots}_{\text{software}} \underbrace{\{b_0, b_1, \ldots, b_i\}\{b_i, b_{i+1}, \ldots, \} \cdots}_{\text{hardware}} \right\}$$

4.2.2. Expected Performance Gain: speedup

- Performance metrics are many:
  - But, We choose "**execution rate**". This is partially motivated by the fact that the performance gain is important and relatively easy to measure.
  - We use profiling information to collect the total execution time as well as the fraction of that time spent in each subroutine.
- Assume :
  - s(i) : the expected execution time of one invocation of the subroutine (basic block) i.
  - h(i) :the expected execution time of subroutine (basic block)i in hardware
  - m(i): the time and cost to of interface between software and hardware for feature I
- **Speedup (γ)**:

$$\gamma = \frac{\text{hardware speed}}{\text{software speed}} = \frac{\frac{1}{\text{hardware time}}}{\frac{1}{\text{software time}}} = \frac{\text{software time}}{\text{hardware time}}$$

- The **speedup from one feature γ(i),** i ∈ C

$$\gamma(i) = \frac{s(i)}{h(i) + \underset{\uparrow}{m(i)}}$$

m(i) is ignored for now

Application Speedup Due to Single feature improvement (i.e., Amhahl's Law)

- Assume for the moment that we just use this single feature in our design.
  - From profiling, the fraction of time spent on the feature is f(i).
- The overall speedup of the application will be:

$$\Gamma = \left[(1 - f(i)) + \frac{f(i)}{\gamma(i)} \right]^{-1}$$

- From this equation, we can observe that increasing the hardware speed of a single feature has less and less impact on the performance of the applications as **its frequency decreases**.
- To increase the overall system performance of the application, we also want to examine **multiple features** that will increase the performance of individual components as well as increase the aggregate **fraction** of time spent in hardware.

The speedup of multiple features in hardware

- We want to evaluate the system gain of a set of features **D** where each member of the set contributes to the system performance based on the fraction of time spent in that feature.
- To estimate the performance of a partition, we can add features and rearrange the terms to get the overall expected performance gain

$$\Gamma(\mathbb{D}) = \left[1 + \sum_{i \in \mathbb{D}} \left(\frac{f(i)}{\gamma(i)} - f(i)\right)\right]^{-1}$$

D is the set of features included in the design

The speedup of multiple features in hardware

- The speed up formula can be simplified

$$\Gamma(\mathbb{D}) = \left[1 + \sum_{i \in \mathbb{D}} \left(\frac{f(i)}{\gamma(i)} - f(i) \right) \right]^{-1}$$

- To the following:

$$\frac{1}{\sum \frac{f_i}{\gamma_i}}$$

Example of Multiple Features Speedup

- No feature is implemented in hardware,
  - speedup=1
- If only feature (1) is implemented in hardware
  - Speedup=1.4
- If features (1) and (2) are implemented in hardware
  - Speedup=2
- If all features are implemented
  - Speedup=2.7

feature	$f(i)$	$\gamma(i)$
1	0.4	4
2	0.3	3
3	0.2	2
4	0.1	1.5

4.2.3. Resource Considerations

- On an FPGA, there are a **finite number of resources** available to implement hardware circuits.
- Thus, given a large number of features, designers are often forced to limit the number of features implemented in hardware based on those that make the most impact.
- Assume:
  - r_{FPGA} that represents the total number of logic cells available
  - r(i) can be used to represent how many logic cells each feature i requires.
- A typical Platform FPGA has multiple types of resources. A vector is a better representation.

Resource Consideration: The resource constraint

- For example, the following is a subset of the available resources in one FPGA:

$$\vec{r}_{\text{FPGA}} = \begin{pmatrix} r_0 \\ r_1 \\ \cdots \\ r_{n-1} \end{pmatrix} = \begin{pmatrix} 11,240 \\ 298 \\ 320 \\ 20 \end{pmatrix} \text{ where } \begin{array}{c} r_0 \text{ is the number of CLBs} \\ r_1 \text{ is the number of BRAMs} \\ r_2 \text{ is the number of DSP Slices} \\ r_3 \text{ is the number of GTX Transceivers} \end{array}$$

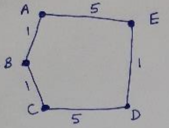
- The **resource constraint equation** is :

$$\sum_{i \in \mathbb{D}} \vec{r}(i) < \vec{r}_{\text{FPGA}}$$

where **D** is the set of features included in the design.

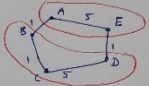
Partitioning of a Design

- Design consists of
  - ***n*** nodes. Each node has an area (typical 1).
  - ***L*** weighted links connecting nodes
- Partition of the design:
  - Goal: partition the design into ***k*** sub-sets with minimum cost function.
  - Link weights in the cost function are counted as follows:
    - External links (between sub-sets) are counted in the cost function.
    - Internal links (within sub-set) are not counted in the cost function.
  - Examples of cost function:
    - $Cost = \sum^{n} ExternalWeights$
    - $Cost = \sum^{n} ExternalWeights + \sum^{k} (SubSet \quad area)^2$



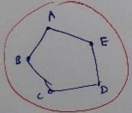
cost = $\sum$ External weight

2 - partitions
cost = 2

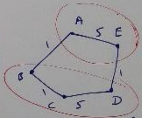


Cost = $\sum$ external weights + $\sum$ area

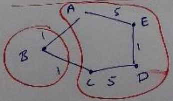
area = (# of nodes)^2



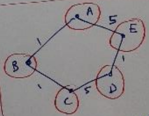
cost = $\underbrace{5^2}_{\text{area}} = 25$



cost = $2 + \underbrace{3^2 + 2^2}_{\text{area}} = 15$



Cost = $2 + 1^2 + 4^2 = 19$

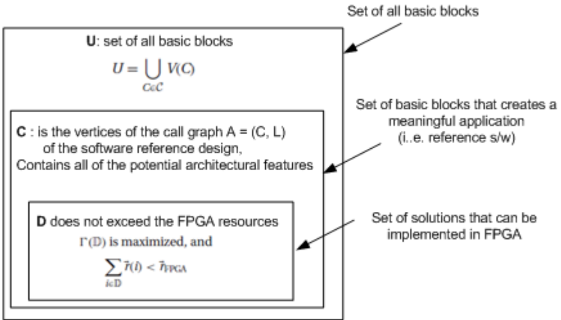


Cost = 13 + 5 = 18

4.2.4. Analytical Approach:

- Problem Statement: find a partition of all of the basic blocks of the application and then separate those sets into hardware and software.
- Formally, we are looking for a partition ***P*** :
  - belongs to the basic blocks U
  - belongs to C where A = (C, L) of the software reference design (contains all of the potential architectural features)
  - belongs to D: does not exceed the FPGA resources.

- We need to find ***P***


Set of all basic blocks
U: set of all basic blocks
$U = \bigcup_{C \in \mathcal{C}} V(C)$
Set of basic blocks that creates a meaningful application (i..e. reference s/w)
C : is the vertices of the call graph A = (C, L) of the software reference design, Contains all of the potential architectural features
Set of solutions that can be implemented in FPGA
D does not exceed the FPGA resources
$\Gamma(\mathbb{D})$ is maximized, and
$\sum_{i \in \mathbb{D}} \vec{r}(i) < \vec{r}_{FPGA}$

Methods to Find Partition *P:* Algorithmically

- Algorithmically:
  - Find all partitions of U, synthesize and profile each partition, and then quantitatively evaluate every Speedup.
  - Unfortunately, number of partitions is huge!
- This means that the number of partitions to evaluate is enormous.

Methods to Find Partition *P:* Heuristic

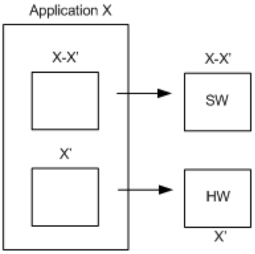
- Heuristic = Trial-and-Error. The steps are:
- Start point:
  - start with the partition provided by the software reference design.
  - That is, we use the subroutines of the original application.
- Create list of subroutines
  - Determine the fraction of time (spent in each subroutine) using profiling tools.
  - List the subroutines in decreasing fractions of time.
  - Focus on those subroutines with a large f.
- Iteratively manipulate the partition X = {X0, X1, . . .} by creating new basic block subsets, merging subsets, and moving basic blocks from subset to subset. The basic idea is to look for changes :
  - **increase the fraction of time "f"** or
  - **increase γ of a potential feature**.

Increasing the Fraction of Time: f(i)

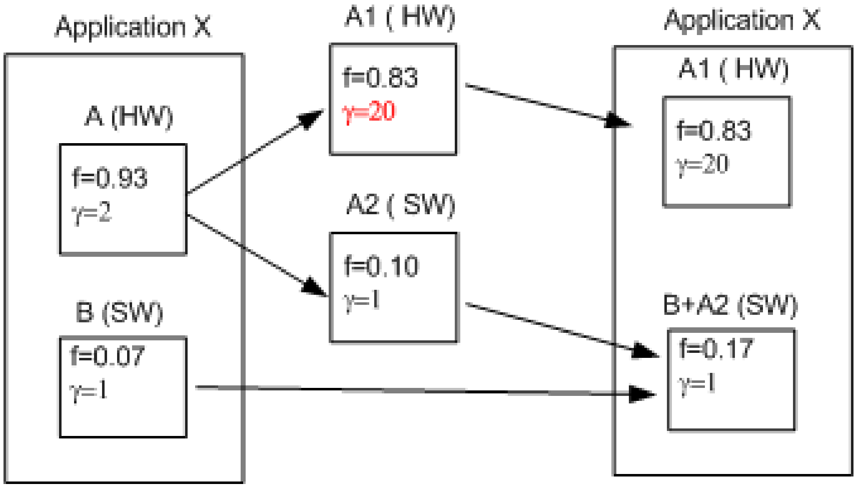
- Increase the fraction of time spent in a subroutine is by making the **feature larger**. This can be accomplished by
  - Initially looking for relationships in the **call graph** or
  - (after the partition has been manipulated) by looking for relationships in the **control flow** graph that bridge subsets.
- The problem of making feature larger:
  - increases the number of resources used
  - may decrease speedup

Increasing the Performance Gain: γ(i)

- To increase the performance gain of a feature, you need to:
  - Examine control flow graph of the feature and
  - Change it to make it more sequential or more parallel
- Making a **feature smaller** has the potential of increasing its performance gain because:
  - Large feature with many basic blocks may increase the sequential behavior of the feature, lowering its γ.
- In terms of the application, this means:
  - taking a subroutine X and breaking it into two subroutines X−X' and X', where subroutine X−X' invokes X'.
  - If X' extracts the parts of X that can be improved in hardware and leaves the sequential parts in X−X', then the γ of X' will be higher than the γ of the original X. Most likely, it will require fewer hardware resources as well.



Increasing the Performance Gain: Example



Speedup(Γ) = 1.869

Speedup(Γ)= 4.739

Summary of the Heuristic method

- Heuristic works by
  - Examining the CG and CFGs of the application and then
  - Making incremental changes to the subset of the partition. Changes are guided by:
    - trying to increase the **fraction of time** spent in a subroutine while not dramatically increasing resources or decreasing performance;
    - Try to **increase the performance** without substantially reducing the fraction of time spent in a subroutine

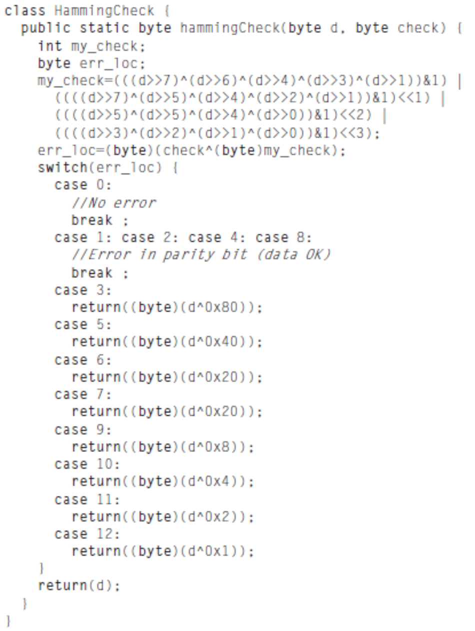
Outline of This Chapter

4.3. Communication

- When a single application is **partitioned** into hardware and software components, it is necessary for those components to **communicate**.
- There number of **issues** related to communicating state across partition boundaries and the **mechanism** for transferring control between hardware and software.

Example: Hamming Check Routine

- Consider the subroutine HammingCheck.
- It involves many simple **bit manipulations** for error-correcting code.
- **Hardware implementation** of this subroutine will substantially outperform the software implementation and require relatively few FPGA resources.
- Hence, The hamming Check is identified as a potential feature and its **hardware implementation becomes a candidate** for inclusion in the final system.



```
class HammingCheck {
  public static byte hammingCheck(byte d, byte check) {
    int my_check;
    byte err_loc;
    my_check=(((d>>7)^(d>>6)^(d>>4)^(d>>3)^(d>>1))&1) |
      ((((d>>7)^(d>>5)^(d>>4)^(d>>2)^(d>>1))&1)<<1) |
      ((((d>>5)^(d>>5)^(d>>4)^(d>>0))&1)<<2) |
      ((((d>>3)^(d>>2)^(d>>1)^(d>>0))&1)<<3);
    err_loc=(byte)(check^(byte)my_check);
    switch(err_loc) {
      case 0:
        //No error
        break ;
      case 1: case 2: case 4: case 8:
        //Error in parity bit (data OK)
        break ;
      case 3:
        return((byte)(d^0x80));
      case 5:
        return((byte)(d^0x40));
      case 6:
        return((byte)(d^0x20));
      case 7:
        return((byte)(d^0x20));
      case 9:
        return((byte)(d^0x8));
      case 10:
        return((byte)(d^0x4));
      case 11:
        return((byte)(d^0x2));
      case 12:
        return((byte)(d^0x1));
    }
    return(d);
  }
}
```

Hamming Check Example: Interaction with Processor

- So, how does this feature (implemented in hardware) and processor **interact**?
- The set of rules that govern this interaction is called the **interface**;
- The interface could be as simple as transferring two bytes to the hardware core, signaling the core to start, and then waiting for the result. This highlights the two main issues of the interface:

1. How is the feature invoked?
2. How is the transfer of state handled?

4.3.1. Invocation/Coordination

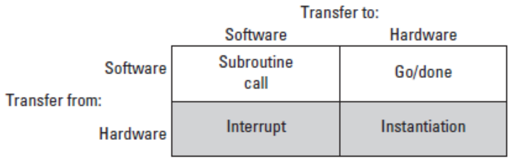
- In software :
  - In sequential computing model: invocation is the **subroutine call**
  - In multi-threading model: Individual threads of control can invoke **subroutines** but also have the ability to fork a new thread of control and different threads **communicate** through a variety of mechanisms e.g., semaphores, monitors, etc.
- Hardware is different from software:
  - In hardware, there is no thread of control, generally, hardware is controlled by some **state** machine that is **always "on".**
  - Statemachines have some sort of "**idle**" state and transfer of control can be thought of as leaving and entering the idle state.
- In general, there are three common approaches to coordinating hardware and software components:
  - Coprocessor model (the most common)
  - Multithreaded
  - Network-on-chip models

Invocation/Coordination : coprocessor model

- This model know as Go/Done or Client/Server. In this model:
  - Hardware sits in an idle state, waiting to provide a service to the processor.
  - The processor signals "**go**" for the hardware to begin and then waits (i.e., blocks itself).
  - When the hardware finished, it signals "**done**," which unblocks the invoking process.
  - The hardware goes back into the idle state, waiting for another go signal.
- A variation on this model has the hardware design doing some work when it is not invoked.
  - For example, the component may be counting external electrical pulses over a period of time to compute the frequency. The go/done interface is only used when the processor needs to know what current frequency is

Invocation/Coordination : coprocessor model

- How to handle the time while the feature is operating:
  - **Fixed Timing**: For small features that take a fixed, short amount of time to complete, the most efficient mechanism is for the processor to simply do an appropriate number of “no-op” instructions and then retrieve the results.
  - **Spin-Lock (or Polling)**: When the amount of time a feature runs is unknown, the processor uses a loop to repeatedly check some condition, indicating hardware has completed its computation.
  - **Blocking (or interrupt)**: Treat the hardware like an I/O device. The “done” signal can be routed as an interrupt to the processor.
- So the four combinations how software and hardware may transfer control.
  - The shaded lower half of the figure shows transfer-of-control mechanisms that are not always supported.



Transfer from:	Transfer to: Software	Transfer to: Hardware
Software	Subroutine call	Go/done
Hardware	Interrupt	Instantiation

Figure 4.3. Transfer of control between hardware and software.

Invocation/Coordination : Other Models

- Mutlithreaded model
  - The processor and hardware components are both running continuously.
  - Coordination is handled by concurrent process techniques:
    - semaphores,
    - messages, and
    - a shared resource such as RAM.
  - A set of system-wide semaphores are used to keep hardware and software threads consistent. Instead of sitting in an
  - “off” state, hardware conceptually sits in a blocked state waiting on a semaphore or for a message.

Invocation/Coordination : Other Models

- Network-on-chip Model
  - This model is based on a completely distributed state and rely on a message-passing to explicitly transfer state.
  - NoC is an IC with multiple modules, which implements a network-based communication between modules.

4.3.2 Transfer of State

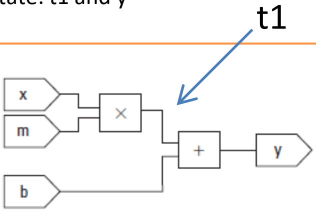
- Maintaining **consistent state** across discrete FPGA devices and processors has been especially **challenging**. Why?
  - Each FPGA device had its own **independent memory** hierarchy,
  - state is **widely distributed** in the design,
  - there is a wide **variety of interconnects** between FPGAs and processors (from slow I/O peripheral buses to high-speed crossbar switches).
- Transfer of State Problem:
  - We need to **explicitly communicate (**between processor and feature) **affected state** that is **trapped.**
- To understand the "Transfer of State" definition, we define two concepts: affected state and trapped state.

Transfer of State: Affected State

- The affected state is:
  - as application data, during a transfer of control, that has the potential of being either **read** or **written** by either the feature or the processor.
  - It is the state we need to keep consistent.
- Affected state: Example 1
  - Affected state: x, m, and b, as well as the return value of the function.
  - Not included in the affected state: t1 and y

```
int trans(int x, int m, int b) {
    int y ;
    y - m*x + b ;
    return y ;
}
```

(a)



(b)

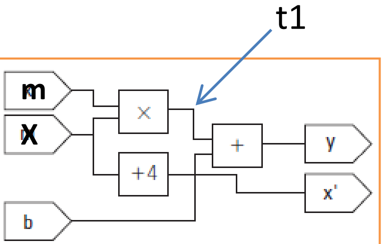
Figure 4.5. Translate.

Transfer of State: Affected State

- Affected state: Example 2
  - Affected state: x, m, and b, return value of the function and **x’**
  - **Note: x is passed by reference**
  - Not included in the affected state: t1 and y

```
extern int m, b ;
int trans_and_inc ( int &x ) {
    int y;
    y = m*x + b ;
    x++ ;
    return y ;
}
```

(a)



(b)

Figure 4.6. Translate and increment.

Transfer of State: Trapped State

- The state of an application is **stored in several parts** of a system:
  - In the processor core alone, application data might reside in
    - a general-purpose register, or
    - a cache, or
    - in a translation look-aside buffer (if an MMU is present and enabled)
  - Outside the processor, application state might exist in
    - on-chip Block RAM, or
    - a subsystem buffer, or
    - external RAM.
- **Trapped state**: it is the state which **resides in non-common location** and it has to be **transferred through the interface**.
  - Example of "Common location": shared memory
  - Example of "non-common location": register inside processor, processor's cache

Examples of where States can be trapped

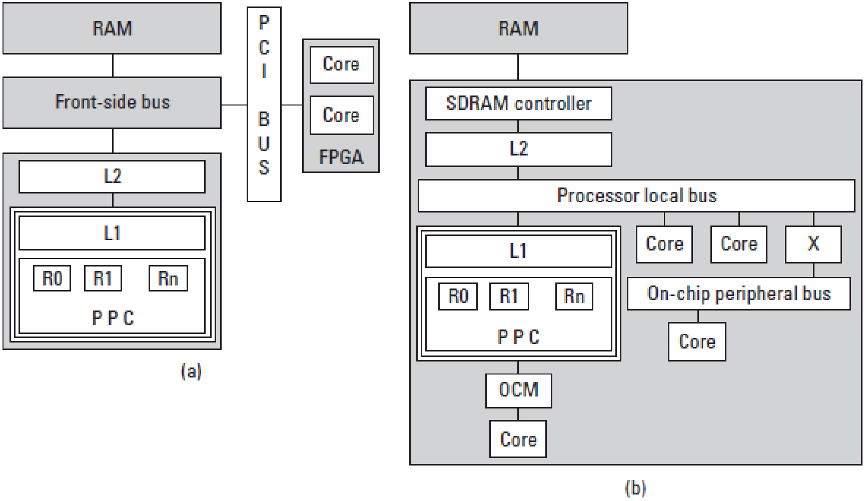


Figure 4.7. Two examples of where state can be trapped (a) traditional and (b) Platform FPGA.

Transfer-of-State Problem

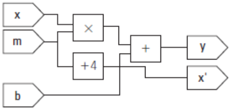
- Transfer-of-State Problem:
  - Affected state that is trapped needs to be explicitly communicated.
  - This is done by a process called marshaling.
- **Marshaling** groups elements of the application's affected state into logical records that are explicitly transferred.
- There are up to four different kinds of records that may be used.
  - Two kinds are used to transfer an initial set of elements to the feature (**Type-I, for initial**) and a final set of elements back to the processor (**Type-F, for final**)
  - These transfers take place when the feature is loaded and unloaded.
  - Another two kinds of marshal records are used repeatedly — one when the feature is invoked (**Type-CI, for copy-in**), and the other when it completes (**Type-CO, for copy-out**).

Marshal Records Examples

- Example of type-CI and type-CO records

```
extern int m, b ;
int trans_and_inc ( int &x ) {
    int y;
    y = m*x + b ;
    x++ ;
    return y ;
}
```

(a)



(b)

Figure 4.6. Translate and increment.

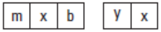


Figure 4.8. Type-CI and Type-CO marshal records.

- Example of a Type-F record
  - suppose a core implemented in the configurable logic accumulates a value in a global variable (it may be statistics about its operation or samples it was given).
  - A typical use of a Type-F record would be to read that variable after the last invocation

Transfer of State Mechanisms

- Mechanisms:
  - **Push**: copy the entire record, stop the processor until the feature completes, and then transfer the entire record back. The caller transmits data before transferring control.
  - **Pull data**: record already exists and is in a known location in memory, then the caller can transfer control and the callee pulls data.
- Transfer can be instantaneous or continuous
  - Instantaneous:
    - Is used when the transfer record is **small**.
    - the feature is designed to reduce the **latency** of the task
  - Continuous:
    - the logical record and the actual data record are **large**.
    - it makes sense to develop a regular access pattern that transmits data continuously.
    - In such a scheme, the feature begins working on part of the transfer record while data are in transit
    - is used when the feature is designed to increase the **throughput**
    - Continuous transfer usually involves setting up **DMA** transfers and the incorporation of FIFO buffers. It can increase the **area** and sometimes increase the latency.

Outline of This Chapter

4.4. Practical Issues

- Numerous **practical caveats and pitfalls** that can confound the analytical approach.
- We will discuss number of these issues **not addressed by the formal** analytical solution.

4.4.1. Profiling Issues

- Profile information is used to approximate the subroutine fraction of time.
- However, a number of situations could generate **misleading results**.
- Data-dependent Execution
  - For some applications (especially when the size of the input data set is changing or when the application responds to external events) a single data set will not be representative for behavior of the application.
  - It is likely that the importance of various subroutines (based on their fraction of execution time)will change with respect to one another.
  - There are a number of ways to address this issue:
    - **Manually analyze** the algorithms and understand the basic operation of the application.
    - Collect profiling information based on **a number of different (size and usage) data sets**.
    - Try to separate the application, perhaps along module boundaries, and profile each module independently.

Profile Issues

- **Correlated Behavior**
  - Some applications are written to specifically incorporate time . Because many profiling systems use an interval timer to sample the program counter, they rely on the application to be statistically uncorrelated to the timer.
  - As a result, the profiler will not provide a statistically accurate description of the system. It could either over- or under-represent a periodic event at the expense of other components in the software reference design.
- **Phased Behavior**
  - During the execution, control moves between clusters of related operations, i.e., the execution exhibits locality.
  - Example: consider an application which :
    - has three routines : A, B, and C. Assume each account for 33% of the execution time (sequentially). The routines are ordered such that A runs to completion before B begins and B completes before C begins
    - If there is only room for one routine in the FPGA resources then one approach is to **time-multiplex** the hardware.

Profile Issues

- **I/O effects**
  - Unfortunately, most profilers do not account for I/O.
  - This means that if the application spends a lot of time waiting for external events, then improving the performance of computationally intense parts of the application may not yield the expected overall system performance gain.
- **Number of Calls:**
  - The number of calls is a major clue to the engineer that perhaps the data set is not describing the behavior of the system accurately.

4.4.2. Data Structures

- Software reference designs are biased toward software implementations.
- Consider the loop examples:
  - both take O(n) steps. However, with hardware, the latter is much more desirable
- There are several places where common software structures can be rearranged to yield better hardware designs:
  - Pointers: Change pointers to “flat” structures (such as vectors and arrays).
    - Flat structures produce regular memory accesses that can subsequently be prefetched or pipelined
  - bit widths: Software programmers generally assume a fixed, standard
    - bit width even though they only use a few bits of information.
    - hardware excels at managing arbitrary bit widths

```
while( i!=NULL ) {
  proc(i) ;
  i = i->next ;
}
```

and

```
while( i<n ) {
  proc(x[i]) ;
  i = i + 1 ;
}
```

4.4.3. Manipulate Feature Size

- In some cases: making the hardware feature smaller and easier to implement
  - The simplest change involves breaking up subroutines into smaller subroutines.
  - This has the effect of isolating computationally intensive parts of the subroutine
- Alternatively, there are times when aggregating subroutines is needed.
  - For example, if two subroutines each account for 25% of the application's time, then they are strong candidates for hardware.
  - However, if implemented individually, each invocation incurs the cost of interfacing

Chapter-In-Review

- described the general problem
- looked at specific profiling data
- formulated an analytical solution
- addressed pragmatic short comings of model

Chapter 4 Terms

partitioning	the process of decomposing an application into multiple implementations (such as hardware and software implementations)
feature	a portion of software that is a candidate for hardware implementation (by candidate we mean the hardware implementation is much faster than software implementation)
interface	is the set of rules that govern this interaction between two entities (for example, the interaction between a processor and an I/O device)

Chapter 4 Terms

marshaling	hardware and software implementations have different views of the state of the machine; marshaling is the process of explicitly organizing the transfer of state between the two
instrumentation	when a compiler or synthesis tool adds extra functionality to a task to help determine its characteristics within an application; for example, tasks are often instrumented to measure the amount of time spent completing the task or how much time is spent idle