Embedded Systems Design with Platform FPGAs

Partitioning

Ron Sass and Andrew G. Schmidt
http://www.rcs.uncc.edu/~rsass
University of North Carolina at Charlotte
Spring 2011
Chapter 4 — Partitioning
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
assuming the software reference model is a sequential program, then **partitioning** is

- the act of grouping sets of instructions,
- mapping each group to either hardware or software,
- implementing the groups destined for hardware in HDL,
- creating the interfaces to transfer control back and forth
Partitioning

Feature

A hardware group or cluster of instructions is called a **feature** once implemented in an HDL and added to a base system, a feature specializes the computer system architecture (hence the name — it becomes an architectural feature)

- also know as a “computational kernel” or just “kernel”
- idea is to solve the general problem by selecting suitable features from the program to be implemented in hardware
- but the big question is “what are the suitable features?”
Partitioning

Profiling

- to answer the question we need data
- simplest way to collect data is to profile the software
- reference design
- gprof is a good, general-purpose profiler
- sample-based profilers, such as gprof, work by
  - interrupting application at fixed time intervals
  - building a histogram from the program counters
  - keeping track of total run-time
- afterwards, these data can be used characterize where “most of the time is spent” in a program
- some profilers also keep track of the number of times a subroutine is invoked
for `gprof`, the program is compiled with the the `-pg` option

after execution, there is a `gmon.out` binary with the profile information

the `gprof` command reads `gmon.out` and gives us a “human readable” version of the profile information in table form

a handy Python script (Gprof2dot) combined with a graph layout application called GraphViz does an excellent job presenting the information graphically
Simple Timing

- the `time` command can be used get a general picture
  
  ```
  % time ./simple
  real    1m14.991s
  user    0m21.959s
  sys     0m0.523s
  ```

- the first number is the “wall clock” time while the second and third represent the time spent computing in the program and operating system, respectively

- the unaccounted for time was waiting for I/O
## AES gprof Output

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>ms/call</td>
<td>ms/call</td>
</tr>
<tr>
<td>93.87</td>
<td>8.12</td>
<td>8.12</td>
<td>14145726</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>5.55</td>
<td>8.60</td>
<td>0.48</td>
<td>27629</td>
<td>0.02</td>
<td>0.31</td>
</tr>
<tr>
<td>0.29</td>
<td>8.62</td>
<td>0.03</td>
<td>1</td>
<td>25.00</td>
<td>25.00</td>
</tr>
<tr>
<td>0.29</td>
<td>8.65</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>8.65</td>
<td>0.00</td>
<td>55282</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>8.65</td>
<td>0.00</td>
<td>93</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>8.65</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>8.65</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>8.65</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>8.65</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>8.65</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>8.65</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>8.65</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative time</th>
<th>seconds</th>
<th>self seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.97</td>
<td>2.42</td>
<td>2.42</td>
<td>20</td>
<td>121.02</td>
<td>121.02</td>
<td>amp1</td>
</tr>
<tr>
<td>16.83</td>
<td>4.82</td>
<td>2.40</td>
<td>20</td>
<td>120.02</td>
<td>120.02</td>
<td>amp2</td>
</tr>
<tr>
<td>16.41</td>
<td>7.16</td>
<td>2.34</td>
<td>20</td>
<td>117.02</td>
<td>117.02</td>
<td>far2</td>
</tr>
<tr>
<td>11.01</td>
<td>8.73</td>
<td>1.57</td>
<td>20</td>
<td>78.51</td>
<td>78.51</td>
<td>amp3</td>
</tr>
<tr>
<td>10.94</td>
<td>10.29</td>
<td>1.56</td>
<td>20</td>
<td>78.01</td>
<td>78.01</td>
<td>far3</td>
</tr>
<tr>
<td>10.66</td>
<td>11.81</td>
<td>1.52</td>
<td>20</td>
<td>76.01</td>
<td>76.01</td>
<td>far1</td>
</tr>
<tr>
<td>4.70</td>
<td>12.48</td>
<td>0.67</td>
<td>2920</td>
<td>0.23</td>
<td>0.23</td>
<td>h_inner13</td>
</tr>
<tr>
<td>2.88</td>
<td>12.89</td>
<td>0.41</td>
<td>1920</td>
<td>0.21</td>
<td>0.21</td>
<td>e_inner13</td>
</tr>
<tr>
<td>2.52</td>
<td>13.25</td>
<td>0.36</td>
<td>1</td>
<td>360.05</td>
<td>360.05</td>
<td>grid</td>
</tr>
<tr>
<td>1.82</td>
<td>13.51</td>
<td>0.26</td>
<td>1</td>
<td>260.04</td>
<td>260.04</td>
<td>tstep</td>
</tr>
<tr>
<td>1.19</td>
<td>13.68</td>
<td>0.17</td>
<td>778</td>
<td>0.22</td>
<td>0.22</td>
<td>asuby</td>
</tr>
<tr>
<td>1.05</td>
<td>13.83</td>
<td>0.15</td>
<td>2</td>
<td>75.01</td>
<td>75.01</td>
<td>initial</td>
</tr>
<tr>
<td>0.98</td>
<td>13.97</td>
<td>0.14</td>
<td>2</td>
<td>70.01</td>
<td>250.04</td>
<td>SYMMLQ</td>
</tr>
<tr>
<td>0.56</td>
<td>14.05</td>
<td>0.08</td>
<td>2334</td>
<td>0.03</td>
<td>0.03</td>
<td>DAXPY</td>
</tr>
<tr>
<td>0.42</td>
<td>14.11</td>
<td>0.06</td>
<td>1560</td>
<td>0.04</td>
<td>0.04</td>
<td>DDOT</td>
</tr>
<tr>
<td>0.42</td>
<td>14.17</td>
<td>0.06</td>
<td>1</td>
<td>60.01</td>
<td>60.01</td>
<td>free_space</td>
</tr>
<tr>
<td>0.35</td>
<td>14.22</td>
<td>0.05</td>
<td>1556</td>
<td>0.03</td>
<td>0.03</td>
<td>DCOPY</td>
</tr>
<tr>
<td>0.14</td>
<td>14.24</td>
<td>0.02</td>
<td>10</td>
<td>2.00</td>
<td>2.00</td>
<td>Je1</td>
</tr>
<tr>
<td>0.07</td>
<td>14.25</td>
<td>0.01</td>
<td>10</td>
<td>1.00</td>
<td>1.00</td>
<td>Jm1</td>
</tr>
<tr>
<td>0.07</td>
<td>14.26</td>
<td>0.01</td>
<td>1</td>
<td>10.00</td>
<td>10.00</td>
<td>rectangle</td>
</tr>
<tr>
<td>0.00</td>
<td>14.26</td>
<td>0.00</td>
<td>40</td>
<td>0.00</td>
<td>0.00</td>
<td>wave</td>
</tr>
<tr>
<td>0.00</td>
<td>14.26</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>tst_params</td>
</tr>
</tbody>
</table>
Partitioning

AES and FDTD Analysis

- the AES application, `aes_encrypt` took 93.87% of the time (very good) but it was invoked 14,145,726 times!
- FDTD is a computational science application
  - six functions take most, but not all, of the execution time
  - however, each function was only invoked 20 times
Amdahl’s Law

- well known in computer architecture

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

- So if only one feature implemented in FDTD:
  20% speedup (max)

- But if top six features implemented in FDTD:
  5.8× speedup (max)
Partitioning

FDTD Gprof2dot/GraphViz

main
100.00%
(0.00%)

amp1
16.97%
(16.97%)
20×

amp2
16.83%
(16.83%)
20×

far2
16.41%
(16.41%)
20×

amp3
11.01%
(11.01%)
20×

far3
10.94%
(10.94%)
20×

far1
10.66%
(10.66%)
20×

probes
7.57%
(0.00%)
20×

tem_y
3.51%
(0.00%)
2×

grid
2.52%
(1.82%)
1×
tstep
1.82%
(1.82%)
1×
initial
1.05%
(1.05%)
2×

h_inner13
4.70%
(4.70%)
2920×

e_inner13
2.88%
(2.88%)
1920×
solve_y
3.51%
(0.00%)
2×

SYMLQ
3.51%
(0.98%)
2×

asuby
1.19%
(1.19%)
778×
DAXPY
0.56%
(0.56%)
2334×
Another Graphical Example

Partitioning
Interpretation

- armed with this information, the system designer can make some decisions
- easy case: JPEG2000
  - 11% reading source image
  - 89% encoding
- in other cases though...
  - no computation dominates
  - limited FPGA resources
  - resources/feature varies
  - performance gain/feature varies

some analysis gets us half-way there; then address the pragmatic concerns
Basic Definitions

- an application consists of collection of subroutines that are related via a (static) call graph
- that is,
  - $C_i = (B_i, F_i)$ is a Control Flow Graph (CFG) that represents a subroutine
  - and the call graph is a graph of CFGs:
    $$A = (C, L)$$
    where $L \subseteq C \times C$
- on the surface, it might seem easier to use a less complicated model — say, two groups (the hardware and software) — but this does not allow us to reason about discontinuous regions of hardware features
Formally, we can define a partition of the *basic blocks* as follows.

- let $\mathcal{S} = \{S_0, S_1, \ldots\}$ of some universal set $U$ of subsets of $U$ such that three conditions are met:
  - $\bigcup_{S \in \mathcal{S}} S = U$
  - $\forall S, S' \in \mathcal{S} \mid S \cap S' = \emptyset$
  - $\forall S \in \mathcal{S} \bullet S \neq \emptyset$

- in English, this says that every element in the universe appears in some subset, every element appears in only one subset, and that there are no empty subsets
Partition Example

The universe of $U = \{a, e, i, o, u, y\}$ could be partitioned into two subsets,

$$\mathcal{X}_a = \{\{a, e, i, o, u\}, \{y\}\}$$

which is visually represented by:

![Diagram showing partitioning of the universe U]
Partitioning An Application

the same concept can be applied to an application

- the universe is the collection of basic blocks
- a *natural* partition groups the basic blocks by the subroutine they belong to

\[
S = \left\{ \begin{array}{l}
\{b_0, b_1, \ldots, b_i\}, \{b_i, b_{i+1}, \ldots\}, \ldots \{b_j, b_{j+1}, \ldots\}
\end{array} \right\}
\]

- our job here is to reorganize contiguous basic blocks such that some are destined for hardware

\[
\mathcal{X}' = \left\{ \begin{array}{l}
\{b_j, b_{j+1}, \ldots\} \{b_k, b_{k+1}, \ldots\} \ldots \{b_0, b_1, \ldots, b_i\} \{b_i, b_{i+1}, \ldots\}
\end{array} \right\}
\]

**software**

**hardware**
Now, we can define expected performance gain

- first for the individual features
- then collectively for the system

This will give us a tool to weigh decisions and compare system choices.
Defining Performance

- as we know, performance is not scalar
- rate-of-execution is common measure
- but power, cost, and others are important to embedded systems

Here, we will use rate-of-execution to demonstrate the concepts.
There are four mechanisms for transferring control between hardware and software.

<table>
<thead>
<tr>
<th>Transfer from:</th>
<th>Transfer to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>Hardware</td>
</tr>
<tr>
<td>Subroutine call</td>
<td>Go/done</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Instantiation</td>
</tr>
</tbody>
</table>

**Partitioning**

**Transfer of Control**
Partitioning

Cycles in Call Graph

![Call Graph Diagram]

- main
- gensurf
- sar
- feature
- print
- list
- daxpy

Embedded Systems Design with Platform FPGAs
extern int m, b;
int trans_and_inc ( int &x ) {  
   int y;  
   y = m * x + b;  
   x++;  
   return y;  
}

int trans(int x, int m, int b) {  
   int y;  
   y = m * x + b;  
   return y;  
}
Trapped State

— Many places in the memory hierarchy where data may reside
— also known as serialization, it creates a temporary record to transfer data between hardware and software
Partitioning

Performance

- Transfer speeds can be highly data dependent
- DMA is usually best for large transfers
- However, many features only need small transfers
**Performance Quantified**

- Application/OS/Hardware interfaces
- Mmap avoids system call but does not allow the device to be shared

![Transaction speed by method](image-url)
Chapter-In-Review

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

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)
Partitioning

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