Function Unit Specialization through Code Analysis

Daniel Benyamin and William H. Mangione-Smith

Electrical Engineering Department, UCLA, Los Angeles, CA, 90095
{benyamin,billms}@icsl.ucla.edu

Abstract

Many previous attempts at ASIP synthesis have employed template matching techniques to target function units to application code, or directly design new units to extract maximum performance. This paper presents an entirely new approach to specializing hardware for application specific needs. In our framework of a parameterized VLIW processor, we use a post-modulo scheduling analysis to reduce the allocated hardware resources while increasing the code's performance. Initial results indicate significant savings in area, as well as optimizations to increase FIR filter code performance 200% to 300%.

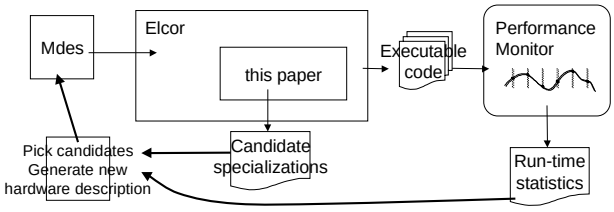

Figure 1: The system overview.

1 Introduction

Digital signal processing (DSP) has exhibited remarkable growth in many mainstream commercial applications, and as a result, now challenges traditional design flows to meet the markets' needs. While these design approaches can take many paths, the usual choices are either the hardware design of an ASIC or the software design for a programmable DSP.

Taking an ASIC approach provides unmatched performance and cost benefits. Unfortunately, ASICs by definition tend to target a narrow range of applications. On the other extreme, programmable processors have become more powerful to meet the growing computational demand. Doing so while maintaining programmability, however, usually produces devices too costly and inefficient. A recent trend is to consider Application-Specific Instruction set Processors (ASIPs) [2]. ASIPs provide dedicated resources for critical tasks (e.g. motion estimation), while offering a complete instruction set for flexibility. The motivation for employing ASIPs is clear. The compiler has a good view of the application at hand, and should utilize the hardware accordingly.

Our approach is to allow the compiler to have complete control over the architectural design of a VLIW processor, a framework which we call Application Specific Configurable Architecture. Within this framework lies techniques to specialize the VLIW function units through code analysis. In this paper, we focus on optimizing loops, a technique which often yields the greatest performance gains in DSP applications. More specifically, we strive towards two independent goals:

1. Create function units for smaller hardware at the cost of run-time performance.
2. Create function units for faster run-time performance at the cost of larger area.

The key to our approach is to use *modulo scheduling* [3] as a guide. From the modulo schedule tables we can decide what functionality is necessary in the datapath, and what operations should be optimized for greatest reduction in cycle count. In contrast to previous efforts, this approach allows for fast and accurate specialization.

2 Related Work and System Overview

Most prior research efforts either attempted to generate code for complex architectural features, or to directly synthesize these features from a base instruction set. Liem [4] took the traditional approach of instruction-set matching, but included techniques to target specialized datapath units. One problem with this and related methods, as seen later, is that more complex functions are less likely to be used. Razdan [1] and Choi [5], on the other hand, both investigated methods of synthesizing new architectural features. Unfortunately, Razdan's methods were limited to optimizing Boolean operations, with little capability of further performance gains. Choi generated complex, multi-cycle units, but did not account for the possible explosion in hardware cost.

Our system augments existing VLIW compiler technology; Figure 1 illustrates the system we are using. The **MDES** and **ELCOR** components are part of the Trimaran compiler system [7, 8, 9]. **MDES** is a flexible machine description system well-suited to VLIW architectures, and **ELCOR** is Trimaran's back-end compiler; it is within the **ELCOR** tool that we have inserted our algorithms. Currently, candidate specializations are produced in a separate file, but no feed-back loop exists. We plan to add the capability of iterating through the design space. All specialization studies take place *after* scheduling, and only "pinhole" optimizations are necessary to target new functionality.

0-7803-5832-X /99/$10.00

3 The Modulo Scheduling Framework

For a given architecture and data/control dependency graph of a software loop, it is possible to find the minimum iteration interval (MII) for a modulo schedule [3]. This lower bound on the length of each loop iteration must be the greater of two bounds, RecMII and ResMII.

ResMII is the minimum schedule length based on resource utilization. As a simple example, if an arithmetic unit performs two additions, and each addition requires two cycles, then $ResMII = 4$. If two arithmetic units were provided, then $ResMII = 2$.

The RecMII is the minimum schedule length based on recurrence dependencies in the graph; such dependencies pass between the iterations of a loop. To characterize these dependencies, a circuit in the graph can be annotated with latency and distance values. The latency of a circuit is the number of cycles required to complete each operation in the graph, and the distance of a dependency between two operations is the number of loop iterations that separate them. Thus, the distance of a circuit is the sum of all distances of the operations in the circuit. Once all circuits have been identified, the RecMII can then be calculated as

$$RecMII = \max_{\forall circuits\ c} \left(\frac{latency\,(c)}{distance\,(c)} \right). \qquad (1)$$

It is important to note that the MII, whether it be defined by the ResMII or the RecMII, may not be possible to achieve [3].

The framework provided by modulo scheduling can alleviate some of the problems faced in specializing function units to the high-level code. First, the schedules only cover loops, and the associated data structures tend to be small. This factor helps reduce the algorithms' runtime. Second, we can use the results of the MII calculations as a guide towards specialization, i.e.:

- *If MII=ResMII, then provide more resources.* In this case the resources are constraining the length of the schedule. The data and control dependencies are such that instruction level parallelism (ILP) is available, but there are not enough resources to perform parallel computations.
- *If MII=RecMII, then reduce the operations' latency.* This approach corresponds to equation (1), where reducing the latency of the operations directly reduces the RecMII. We will see shortly how to determine which circuit achieves the maximum in (1).

4 Specializing for Area

Specializing for area is based on the following approach: each ALU resource (e.g. an integer or a floating point ALU) only uses a fraction of it's hardware in a section of code. We analyze the usage for each resource, and remove functionality when not needed. Since loops consume the bulk of the software run-time, it is the instructions in these loop schedules which have the greatest weight in the system. An algorithm for specializing function units for minimum area now follows:

```
N=num_ALU_resources;
for(i=1 to K scheduled operations){
      Create a new function unit r(i) based
      on an operation and/or operands;
}
while(K>N){
      r(i) = merge(r(i), r(j)) such that
      cost(all r) is minimized;
      if(merge successful){ K=K-1; }
}
```

As an example, Figure 2 shows K=4 instructions scheduled onto N=3 function units. For each instruction, create a resource dedicated to the instruction's datapath. So a `mpy reg, #4` instruction would correspond to a constant coefficient multiplier (or in this case, as simply a shift of two). Since we have only been allotted three function units, we must merge a pair of instructions so that the total area cost is minimized. At worst the cost remains the same (no resource sharing), and at best the cost reduces by the size of one custom function unit (full resource sharing).

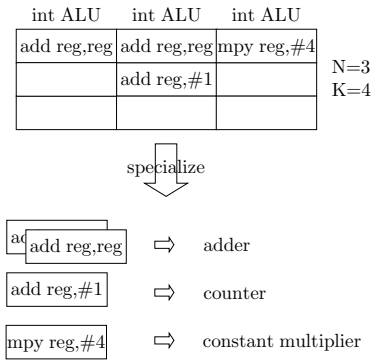

Figure 2: Deducing hardware requirements from a schedule.

Clearly, the heart of the algorithm is the $merge$ function. This function should include a set of heuristics that can make intelligent decisions regarding the merging of different arithmetic operations. For our experiments, we used a simple set of rules:

- If `r_i==r_j`, then merge.
- Merge constant functions with variable functions.
- Keep constant multiplier functions separate.

With these rules we analyzed several loop-intensive programs, such as FIR filters, matrix-mutliplies, and number factoring. Area calculations for various ALU functions were estimated from ALU descriptions of several processor designs. Some area estimates are (relative to an integer or floating point ALU size of 1.0):

int. multiplier: 0.4

int. adder: 0.1

fp. multiplier: 0.58

fp. divider: 0.29

The analysis was performed on a VLIW machine with 4 integer ALUs, 2 FP ALUs, 2 load/store units, and one branch unit. The results are shown in Figure 3 for integer code and in Figure 4 for floating point code.

Clearly, only a small amount of functionality within an ALU resource is used for any particular loop. In many cases, the code required only counters to increment array indices, and one arithmetic operation for the loop body. Note that the benchmarks were done on a machine with 4 integer ALUs, so that on average the integer code required only about one tenth of the hardware area allotted to it (400% vs. 40%). One must be careful, however, to make sure that code *outside* the loop body is able to run as well, so it may be

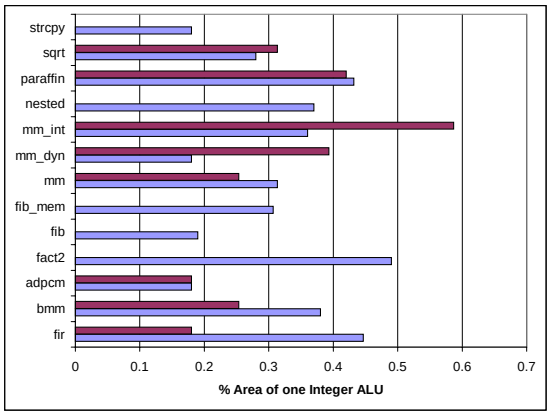

Figure 3: Area requirements for integer code as a fraction of the area of one integer ALU. Programs with multiple bars indicate reduction for multiple loops in the code.

sensible to include one complete ALU and several area specialized ALUs.

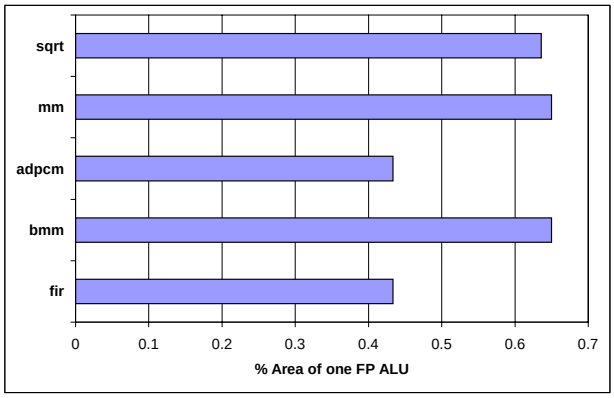

Figure 4: Reduced ALU size for floating point code.

5 Specializing for Performance

We now consider an approach to optimize performance of function units. As mentioned previously, if a loop is coded such that it becomes recurrence constrained, then one must optimize the timing of the operations that define the II. Doing so may be difficult, since the structure of the dependency graph and the timing of a machine's instructions may not indicate the problem.

We define the *critical path* of a loop body to be the circuit c which achieves the maximum in (1). For notation, let op_i be an instruction *i* in the loop body, where op_i is scheduled later than op_{i-1}. Let $schedtime(op_i)$ be the time slot the instruction is scheduled in, and $latency(op_i)$ the latency of the instruction.

Given a modulo-scheduled loop, the following algorithm will usually[1] find the critical path:

```
   choose last scheduled instruction op_i
   that is not a branch;
   for(each precedent instruction op_j of op_i,
j<i){

if(latency(op_j)=schedtime(op_i)-schedtime(op_j)){
        save op_i in critical_path;
        i=j;
     }
   }
```

As an example, the schedule in Figure 5 is for the inner loop of a floating point FIR filter code, with arrows indicating true dependencies, and edge labels indicating the latency of the source instruction. Note that the last two instructions are scheduled at a time greater than the II of 9, and are "wrapped" around the initial nine cycles. The very last instruction is a branch instruction, which will always be the case in a modulo scheduled loop. Thus the algorithm starts with the `st` instruction and traces backwards. When evaluating the inclusion of the `fadd` instruction's predecessors, the algorithm does not choose the `ld` instruction at time 2. This is because the `ld` instruction has a latency of 2 but is allowed 3 cycles before the dependent `fadd` instruction executes, and thus is not considered to be part of the critical path. Indeed, the actual critical path (correctly found by the algorithm), is `ld` → `fmpy` → `fadd` → `st`.

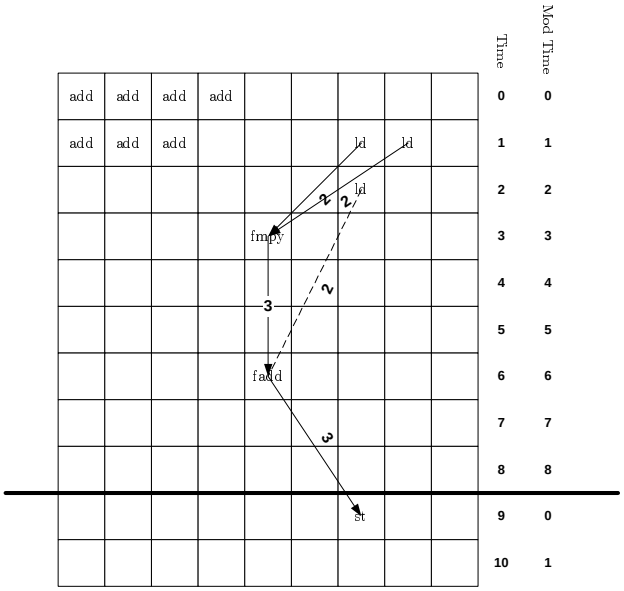

Figure 5: Schedule for FIR filter code, where columns represent hardware resources. Only the first cycle of multi-cycle operations are shown.

Several example programs were run through the algorithm to discover the critical paths, the results of which are documented in Table 1. From these critical paths, the designer (or automated tool) can focus on a function unit which performs a sequence of operations faster than the original code sequence. One common candidate in DSP systems is the `mpy-add` instruction, which Table 1

[1] The algorithm has found the critial path for all of the code tested for this report, but is not guaranteed to do so.

also indicates. However, it is important to note the other sequences of instructions that were discovered, such as `add-add` and `mpy-sub-divide-sub`.

Code Name	Loop Body	Critical Path	Length	II
mm	`s1+=a[i][k]*b[k][j];`	add fl fmpy	6	3
fir	`y[i1]+=w[i2]*x[i1+i2];`	add fl fmpy fadd fs	10	9
sqrt	`x[i] = x[i] - (x[i]*x[i] - (i+1))/ (2*x[i]);`	add fl fmpy fsub fdiv fsub fs	22	20
wave	`a[i][j] = a[i-1][j] + a[i][j-1] + a[i-1][j-1];`	add fl fadd fadd fs	11	9

Table 1: The critical paths for different applications. "Length" refers to the critical path length, which is usually greater than the II (if not, then there is no software pipeline).

5.1 Case Study: FIR Filters

The application of FIR filters, which uses the common multiply-accumulate structure, deserves some focused attention. A faster multiply-accumulate does not necessarily mean a faster FIR filter. For example, the code segment from "mm" in Table 1,

```
s1 += a[i][k]*b[k][j];
```

uses a floating-point multiply and add instruction, which take 6 cycles total to compute. The II for this loop body, however, is just 3 cycles. Optimizing **both** the floating point multiply **and** the add will reduce the II, but combining them into a single instruction will not.

The code in the "fir" example is almost identical algebraically, except that there is some amount of pointer arithmetic to index the array. This arithmetic causes one extra cycle for each calculation, but doing so offsets the modulo schedule. As a result, the amount of ILP in the code is greatly reduced, forcing "fir" to run three times slower than "mm". The clue for optimization here is to combine the pointer arithmetic into the load instruction. By doing so, the "fir" code will see $2\times$ or $3\times$ reduction in cycle count.

6 Conclusions

We have shown a novel way of specializing function units for both area and speed improvements that is computationally simple yet very effective. In a certain sense, we are able of generating entire datapaths, such as FIR filters, for a VLIW architecture. Because we are doing analysis after scheduling, we can guarantee whether a certain specialization will improve performance. Lastly, the modulo scheduling framework tells us whether parallelism exists, but is limited by resource usage, or that high latency resources are inhibiting parallelism in the code.

References

[1] R. Razdan and M.D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *International Symposium on Microarchitecture*, 1994. pp. 172-80.

[2] C. Lee, J. Kin, M. Potkonjak, W. Mangione-Smith, "Media architecture: general purpose vs. multiple application-specific programmable processor," in *35'th DAC*, 1998. pp. 321-6.

[3] B. R. Rau, "Iterative modulo scheduling," in *International Journal of Parallel Programming*, vol.24, Feb. 1996. pp. 3-64.

[4] C. Liem, T. May, P. Paulin, "Instruction-set matching and selection for DSP and ASIP code generation," in *EDAC*, 1994. pp. 31-7.

[5] H. Choi, I.C. Park, S.H. Hwang, C.M. Kyung, "Synthesis of application specific instructions for embedded DSP software," in *ICCAD*, 1998. pp. 665-71.

[6] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *SIGPLAN*, 1988. pp. 318-28.

[7] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. F. Warter and W. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," in *Proc. 18th Ann. Int'l Symposium on Comp. Arch.*, pp. 266-75.

[8] S. Aditya, V. Kathail and B. R. Rau, "Elcor's Machine Description System: Version 3.0" Tech. Report HPL-98-128 (R.1). H-P Laboratories, October 1998

[9] V. Kathail, M. Schlansker and B. R. Rau. "HPL-PD Architecture Specification: Version 1.1" Tech. Report HPL-93-80 (R.1). H-P Laboratories, Sept. 1998.