University of California
Los Angeles

A Study On

The Impact Of Compiler Optimizations On High-Level Synthesis

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Raghu Prabhakar

2012

Abstract of the Thesis

A Study On

The Impact Of Compiler Optimizations On High-Level Synthesis

by

Raghu Prabhakar

Master of Science in Computer Science

University of California, Los Angeles, 2012

Professor Jason Cong, Chair

High-level synthesis is a design process which takes an UN-timed, behavioral description in a high-level language like C and produces register-transfer-level (RTL) code that implements the same behaviour in hardware. In this design flow, the quality of the generated RTL is greatly influenced by high-level description of the language. Hence it follows that both source-level and IR-level compiler optimizations could either improve or hurt the quality of the generated RTL. The problem of ordering compiler optimization passes, also known as the phase-ordering problem, has been an area of active research over the past decade. An optimization has enabling and disabling effects on other optimizations, and such effects are caused by either the nature of the optimization itself, the input program being optimized, or the target platform for which the code is being optimized. A well-known fact in literature is that the standard optimization order chosen by the compiler writer may not be the best order for every input, and hence can end up producing inferior code.

All methods mentioned above are targeted towards compilers producing code that will be executed on a processor. In this study, we explore the effects of both source-level and IR optimizations on high-level synthesis. The parameters of the generated RTL are very sensitive to high-level optimizations, in the sense that a right choice can provide significant

benefits and a wrong choice can cause significant degradation. We consider three source-level optimizations commonly used in High-level synthesis. We study them in isolation and then propose simple yet effective heuristics to apply them to obtain a reasonable latency-area tradeoff. We also study the phase-ordering problem for IR-level optimizations from a HLS perspective. As many optimizations that are employed in a typical HLS flow were originally developed with a standard compiler in mind, and given the increasing popularity of HLS, we feel that such a study is essential to building high-quality HLS tools. Our initial results show that an input-specific order can achieve significant reduction in the latency of the generated RTL, and opens up this technology for future research.

The thesis of Raghu Prabhakar is approved.

Glenn Reinman

Todd Millstein

Jason Cong, Committee Chair

University of California, Los Angeles

2012

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to take this opportunity to thank my adviser, Professor Jason Cong for all his support and encouragement over the course of two years. It is inspiring to have such an adviser who is so passionate about what he does. I am lucky to have had the opportunity to work with a very diverse group of bright researchers in our lab who have been gracious and patient enough to answer all my stupid questions. I would also like to thank Professor Todd Millstein for all the support and guidance. I am grateful to him for taking time off his schedule to meet me or to reply to my insane ideas through clear, detailed responses. The encouragement has always been there, and perhaps that is what has driven me into completing this thesis.

I would like to thank my parents who have always been supportive for everything. I would like to thank my flat-mates and friends without whom masters would not have been as much fun. While the list is endless, I would like to conclude by thanking UCLA for making my masters an experience.

CHAPTER 1

Introduction

The field of compiler optimizations has been an area of active research for more than fifty years. Numerous optimizations have been proposed and deployed over the course of time, each trying to optimize a certain aspect of an input program. With the increasing level of abstraction in programming languages and constantly evolving architectures, newer opportunities for optimization were created. Today, a great deal of literature is available to understand the mechanics, goals and theoretical complexities of each optimization. Optimizations play a key role in evaluating a compiler. Moreover, as processor clock speeds no longer increase in accordance with Moores law, the free ride of better performance obtained every couple of years by increased clock speeds is no longer available. Current trends in architecture favor a heterogeneous mix of general-purpose CPUs, application-specific accelerators like GPUs, FPGAs and hybrid memory hierarchies. Software must be explicitly written to take advantage of such a heterogeneous mixture of computing power. This places a heavy responsibility on the compiler to intelligently transform input code to achieve a certain objective, like code size, execution time or energy.

Given the significance of compiler transformations, it is important to understand the nature of each transformation so that it can be used intelligently. A well-known fact in literature is that optimizations have enabling and disabling interactions among themselves. This problem, also known as the phase-ordering problem, basically implies that applying transformation a before another transformation b can potentially increase the benefit obtained from b or can destroy the same. The interaction that exists between instruction scheduling and register allocation is a classical example for such a behavior. To make matters even more complicated, most of such interactions are input-dependent; depending on the existence of a

certain structure and opportunities in the input program, interactions may or may not exist, and can vary in degree. Unfortunately, studies on most of the transformations have been conducted in unison, without considering the presence of other optimizations. A typical production compiler like GCC has more than 100 optimization flags. Standard optimization orders (i.e., -O2, -O3 etc.) are chosen by the compiler writer based on an averaged performance number on a large benchmark set. Past research has shown that the standard sequence yields inferior code, and that the best sequence is largely dependent on the input, the target architecture and the objective (time, code size etc.). As the solution space is huge, compiler researchers have tried a plethora of methods over the past decade based on searching techniques ([10] [9] [17] [16], [4]), analytical models ([29], [33], [32], [26], [31]), empirical approaches based on statistical data ([25], [24] [2]), and a mixture of all of these ([12], [27], [22]).

However, it is to be noted that all aforementioned approaches have been used to arrive at an optimal optimization sequence that can transform an input program such that the transformed version executes efficiently on a given processor, for some definition of efficiency. In this case, decisions regarding optimization orders are implicitly or explicitly influenced by code execution parameters such as the processor pipeline, size of the instruction window, live variables affecting register pressure, presence of hardware-managed caches and so on. How different would such optimization orders be if the code being optimized was not meant to be executed on hardware? In other words, what is the impact of the order of compiler transformations if the input is a behavioral description of some hardware itself?

For instance, consider the example design in Figure 1.1. The function walks through all elements of array a in a loop and computes the sum and product of all array elements. Note that the loop trip-count and array size are statically known. Array sizes are required to be known at compile-time for HLS to work, and loop trip-count needs to be known statically so that latency estimation can be performed. Also, let us consider a set of three optimizations mentioned below.[1]

[1]We use the letters within brackets to refer to the respective optimizations in this section.

```
int add(int a[20], int o[2])
{
    int i;
    o[0] = 0;
    o[1] = 1;
        for(i = 1; i < 20; i++) {
            if(a[i]%2 == 0) {
                o[0] += a[i];
                o[1] *= a[i];
        }
    }
}
```

Figure 1.1: Example design optimized with -gvn, -indvars and -mem2reg. CPU best sequence differs from HLS best sequence.

- -gvn (g): Global value numbering.
- -mem2reg (m): Promote memory to register.
- -indvars (i): Canonicalize induction variables.

Table 1.1: Benchmarks and description.

Sequence	CPU (cycles)	HLS (cycles)
gim	3903	12
img	3814	22

Consider two sequences, *gim* and *img*. In order to measure and compare the impact of the two sequences on the CPU and HLS, we optimized the design in Figure 1.1 using the two sequences, and ran a final dead-code elimination pass. To measure CPU performance, we compiled the optimized function into a binary by attaching some driver code. We used a cycle-accurate state-of-the-art out-of-order processor simulator Simics to get accurate CPU

performance. We used xPilot to get latency measurements of the physical circuit this design represents, given a target clock period of 5 ns. Table 1.1 summarizes the performance of the sequences in these two scenarios.

We can observe that the sequence *gim* has a better performance in the HLS setting while sequence *img* has a better performance for code executed on the CPU. Upon examining the effects of the optimizations, we find that *img* produces smaller code with fewer loads. This is because the *-gvn* optimization applied after *-mem2reg* is exposed to far more opportunities like re-calculating array indices after their base addresses have been promoted to registers. This explains why the CPU shows better performance on the *img* optimization. However, this raises the question as to why the design with greater number of loads has a smaller latency in the HLS setting, especially because loads are expensive operations. Further examination revealed that while *img* reduced the loads by re-using computed values, it increased the length of the data dependency chain. This led to the *img* design having one extra state in its finite state machine created during scheduling.

The simple example shown above demonstrates that there are very subtle details and side effects that can have different impacts on CPU codes and HLS designs. Also, the impact of one optimization can be more pronounced in a HLS setting than in a CPU design. A typical CPU has many hardware features that enhance the performance of code that is being executed. For example, multiple levels of caches and out-of-order execution drastically reduce the cost of a single load. Techniques like branch prediction and speculative execution can hide the cost of evaluating a branch most of the times in case of loops. High-level synthesis is a different area in that way where each load corresponds to a read from a memory block, and each load costs the same number of cycles. Every branch instruction is dependent on another instruction that computes the exit condition, and branch prediction mechanisms have to be specified in software manually by the designer if needed. Also, HLS can potentially exploit greater ILP limited only by the physical resources available on the target platform. On a typical processor, only the ILP available within the instruction window is exploited.

In this study we perform an initial investigation into the impact of compiler transformations on behavioral descriptions of hardware circuits, which are passed through a high-level

synthesis (HLS) tool. High-level synthesis is an automated design process that takes an UN-timed, behavioral description of a circuit in a familiar, high-level language like C, and generates a timed or UN-timed register-transfer-level (RTL) net-list that implements the same behavior. The generated RTL can then be passed through vendor-specific logic synthesis tools to obtain the final net-list. High-level synthesis provides a convenient way for a designer to specify the behavior of a circuit without worrying about low-level timing details. With the increase in the complexity of circuits application-specific processors, ASICs and FPGA-based accelerators, HLS is growing in popularity in the system design community.

In this study, we investigate the effects of compiler optimization orders on the quality of RTLs generated in HLS, for some definition of quality. The RTL generated by a HLS process is heavily influenced by the way the design is specified at the high-level. As a result, researchers have developed several source-level optimizations, some of them specific to HLS like array partitioning. Each high-level optimization can be tuned using specific parameters, and the choice of the parameters can either enable or disable another optimization. For example, a wrong array partition factor can be detrimental to loop pipelining. We have chosen three important high-level optimizations - loop unrolling, loop pipelining and array partitioning - and examined the interactions between them. We use AutoESL, an industry-standard High-level synthesis tool from Xilinx along with other EDA tools from Xilinx to analyze the effects of the optimizations under study. We describe and use a set of simple, yet effective heuristics to quickly search the space of the described optimizations and study their effects on several benchmarks.

We also study the impact of classical scalar optimizations on High-level synthesis. As AutoESL is a commercial tool that performs many optimizations internally, AutoESL is not a suitable tool to study such lower level optimizations. We motivate our study of this problem using data from xPilot [5], a research High-level synthesis tool developed at UCLA. xPilot provides the flexibility to enable and disable specific low-level optimizations, and thus is an ideal tool for our study. We evaluate several approaches, and suggest a new approach based on *lookahead* for optimizations. Our initial experiments show that latency improvements of more than 3X can be achieved by choosing the right order for an input

behavioural description. We thus highlight the importance of phase ordering and the more general idea of having a strong compilation technology in a HLS tool-chain.

The rest of the thesis is organized as follows. We provide some necessary background information regarding HLS and xPilot in chapter 2. Our study on high-level optimizations are described in chapter 3. For scalar optimizations, we motivate, describe and evaluate all our methods in chapter 4. We discuss related work in chapter 5 and conclude with comments on future work in Section 6.

CHAPTER 2

Background

2.1 High-level Synthesis

High-level synthesis (HLS), or behavioral synthesis, is the process of automatically generating cycle-accurate RTL models from behavioral specifications. The behavioral specifications are typically in a high-level language, like C/C++/Matlab. The generated RTL models can then be accepted by the downstream RTL synthesis flow for implementation using ASICs or FPGAs.

A high-level synthesis tool can be regarded as a compiler for such high-level language, with a target machine that are specifically created to run the particular program. Compared to the traditional RTL-based design flow, potential advantages of HLS include the following.

- Better management of design complexity. Raised level of abstraction leads to reduced code size and thus increased efficiency in design capture, simulation and verification. With the increase of design complexity, this can lead to overwhelming improvement in time to market.

- Code reuse. Since a design is specified using a high-level language without much knowledge about the target platform, it is easy to re-target an existing design towards a different platform or a different frequency. In addition, libraries containing behavioral specifications of common functionalities can be built to facilitate such reuse.

- Easy design-space exploration. By applying different directives (e.g., loop unrolling, memory mapping, pipelining) and constraints (e.g., resource allocation, clock period), different architectures can be generated and evaluated easily. The designer can thus

generate a lot of design alternatives and pick the one that best fits the need of the system.

HLS has been an active research topic for more than 30 years. Early attempts to deploy HLS tools began when RTL-based design flows were well adopted. In 1995, Synopsys announced Behavioral Compiler, which accepts behavioral HDL code and connects to downstream flows. Similar tools include Monet from Mentor Graphics and Visual Architect from Cadence. This wave of tools received wide attention, but failed to widely replace RTL design. This is partly ascribed to the use of behavioral HDLs, which are not popular among algorithm and system designers and require steep learning curves. Since 2000, a new generation of HLS tools has been developed in both academia and industry. Unlike many predecessors, many of them use C-based languages to capture the design. This makes them more accessible to algorithm and system designers. It also enables hardware and software to be specified in the same language, facilitating software/hardware co-design and co-verification. The use of C-based languages also makes it easy to leverage new techniques in software compilers for parallelization and optimization. As of 2012, notable commercial C-based tools include Cadence C-to-Silicon Compiler, Calypto Catapult C (formerly a product of Mentor Graphics), NEC CyberWorkBench, Synopsys Synphony C (formerly a product of Synfora, and originating from the HP PICO project), and Xilinx AutoESL (originating from the UCLA xPilot project [5]). More detailed surveys on the history and progress of HLS are available from [13, 14, 6].

2.2 xPilot

xPilot [5] is an academic HLS tool developed at UCLA. It takes as input a C function and generates an RTL Verilog module to implement the functionality. Internally, the process is divided into the following phases. Compiler transformations are first performed on the source code using LLVM [19] to obtain an optimized intermediate representation, which can be translated to a control-data flow graph (CDFG). Operation scheduling is then performed on the CDFG, and it generates a finite-state machine with data path (FSMD) model, where each

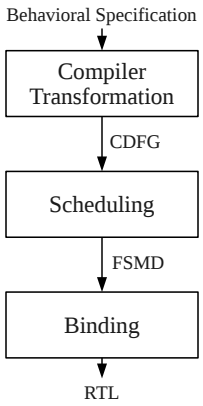

Figure 2.1: The high-level synthesis flow in xPilot.

operation is assigned to a state in the FSM. After scheduling, the cycle-accurate behavior is determined. Binding is then performed on the FSMD to allocate functional units, storage units and interconnects, and then the RTL net-list is decided. The basic flow is illustrated in Figure 2.1. Note that the flow can be organized in slightly different ways, e.g. [7] performs scheduling and binding in an iterative manner.

As mentioned earlier, the LLVM compiler framework [19] is employed as the front-end framework in xPilot. A sequence of LLVM passes, including the standard ones and one specifically designed for HLS, are performed to generate the CDFG. For a given CDFG, the scheduler in xPilot tries to minimize worst-case latency by default, under the constraints of data dependency, control dependency, clock frequency, and resource limits [8]. The scheduler tries to insert clock boundaries on certain edges of the dependency graph, in order to guarantee that the combinational delay of the logic in each state is within the target clock period, and that the number of operations of a certain type (e.g., multiplication) in each state is less than or equal to the number of corresponding functional units (e.g., multipliers). In a simplified model, operations in the same basic block are scheduled into consecutive control states; branches (including loops) are implemented as state transitions in the FSM. Thus, the resulting FSM is somewhat similar to the control-flow graph of the input function. If the control-flow graph of the input function is reducible, it is possible to estimate the worst-case latency of the module given the trip counts of loops.

As shown in Table 2.1, xPilot uses around 55 different optimizations during CDFG cre-

ation . These include standard compiler transformations like dead code elimination, algebraic re-association, common sub-expression elimination, peephole optimizations and so on. There are also certain other transformations which are unique to a HLS-based setting, one example being the *bitwidth minimization* optimization. By default, xPilot applies a pre-determined optimization sequence which has around 250 optimizations. Some optimizations like dead-code elimination are applied many times. In the default setting the user does not have an option to control these optimizations. However, for the purpose of our study, we have modified xPilot so that we have greater control over the optimization sequence used to generate the CDFG. Chapter 4 describes this aspect of our study in detail.

In order to study the effect of IR-level transformations, it is important that we have complete control over all the optimization phases that the code goes through before a CDFG is generated from it. Commercial tools like AutoESL do not provide this flexibility. Although it is possible to perform the transformations, AutoESL performs several transformations upon this input on invocation and hence our analysis would become inaccurate. As explained further in Chapter 3 and Chapter 4, we use AutoESL to study high-level transformations and xPilot to study IR-level transformations.

2.3 Experiment design flow

Figure 2.2 describes the architecture of our flow, which will be described in the subsequent two chapters. Source-level transformations are studied using AutoESL. We use AutoESL's estimates in our iterative approach to quickly arrive at a good configuration for a design, and then invoke the Xilinx back-end. We use xPilot to study IR-level transformations and their impact on the latency of the RTL generated. As we are not factoring area utilization for IR-level transformations into our study here, we do not go through the Xilinx back-end.

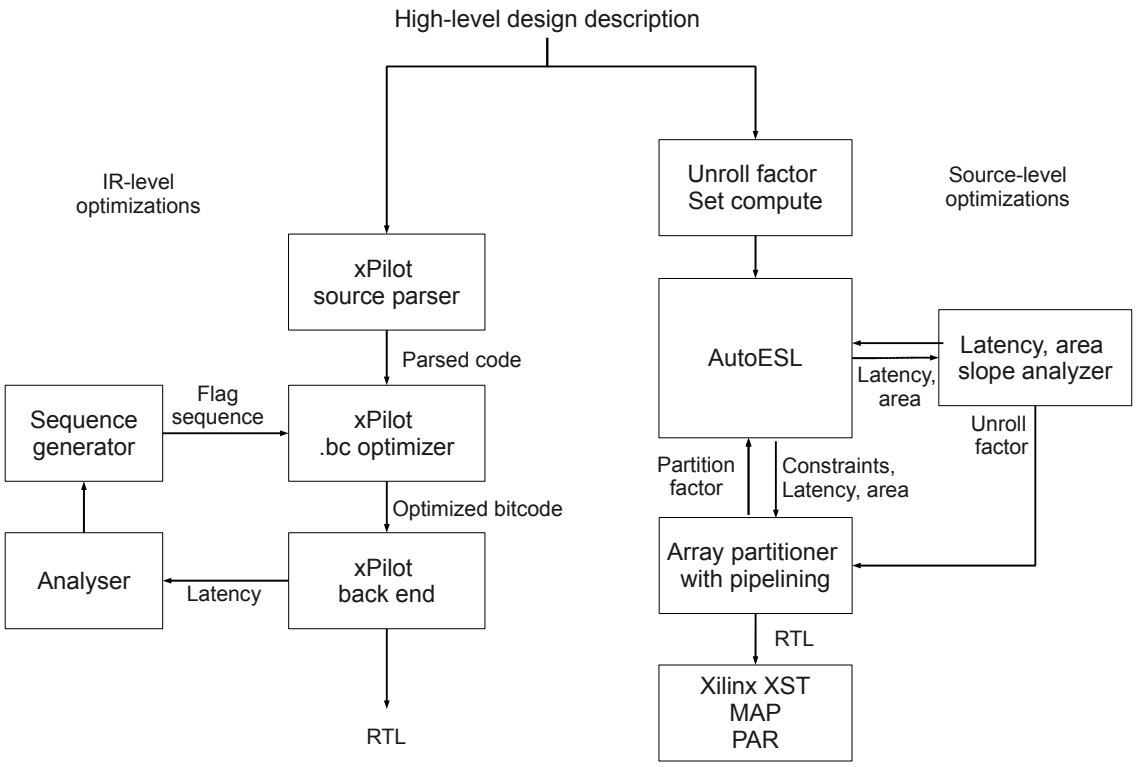

Figure 2.2: Broad design flow used in all our experiments. xPilot is used for IR-level optimizations while AutoESL is used for source-level optimizations.

Table 2.1: Benchmarks and description.

Optimization	**Description**		**Optimization**	**Description**
-ptrLegalization	Convert pointers to array indices		-interface-gen	Interface generation
-func-inst	Instantiate a function		-inst-simplify	Operator strength reduction
-globalsmodref-aa	Mod-ref analysis for globals		-loop-dep	Loop dependency analysis
-bitwidth	Bitwidth analysis		-aggr-aa	Aggressive alias analysis
-flattenloopnest	Flatten a loop nest		-function-uniquify	Uniquify functions
-loop-merge	Fuse loops		-mergereturn	Unify function exit nodes
-mem2reg	Promote memory to register		-array-seg-normalize	Array segment normalization
-constprop	Constant propagation		-globalopt	Global variable optimizer
-doublePtrElim	Eliminate double pointers		-pointer-simplify	Simplify pointers
-directive-preproc	Preprocess directives		-if-conv	If-conversion
-rm-dup-func	Remove duplicate functions		-array-normalize	Normalize array accesses
-argpromotion	Promote 'by reference' arguments to scalars		-array-partition	Array partitioning
-break-crit-edges	Break critical edges in CFG		-loopsimplify	Simplify loops
-dce	Dead code elimination		-loop-deletion	Delete dead loops
-adce	Aggressive dead code elimination		-syn-check	Check synthesizability
-array-map	Perform array mapping		-deadargelim	Dead argument elimination
-function-inline	Function in-lining		-xunroll	Partially unroll loops
-gcse	Global common sub-expression elimination		-ipconstprop	Interprocedural constant propagation
-scalarrepl	Scalar replacement		-func-legal	Legalize function
-ptrArgReplace	Pointer Argment replacement		-deadtypeelim	Dead type elimination
-reassociate	Algebraic re-association		-array-flatten	Flatten arrays into scalars
-indvars	Canonicalize induction variables		-condprop	Conditional propagation
-array-streaming	Specify array as streaming		-instcombine	Combine instructions
-simplifycfg	Simplify the CFG		-expr-balance	Expression tree balancing
-dse	Dead store elimination		-gvn	Global value numbering
-bitwidthmin	Bit-width minimization		-loop-bound	Loop-bound estimation
-licm	Loop-invariant code motion		-basicaa	Basic alias analysis
-simplify-libcalls	Simplify library calls			

CHAPTER 3

Source-level Optimizations

In this chapter, we describe our study of high-level optimization interactions. By high-level optimizations, we are referring to optimizations that are typically performed on a high-level representation of a program like an abstract syntax tree (AST). In this study, we consider three optimizations commonly used on High-level synthesis designs - array partitioning, loop unrolling and loop pipelining. We have chosen these optimizations as they are most commonly employed in standard high-level designs. A quick survey of accelerators in the Open Accelerator Repository [1] shows that these three optimizations are applied in almost every design. AutoESL provides a convenient interface to specify these optimizations either via a directive in a configuration file or as a pragma in the source code itself. We first describe AutoESL's optimization options and study each of the three optimizations mentioned above individually. We then analyze interactions, draw conclusions and describe a set of simple heuristics to quickly eliminate bad choices and arrive at a good configuration.

3.1 AutoESL optimization options

Table 3.1 describes all the directives that AutoESL exposes to users to optimize the design. As can be seen, the solution space is large, not just because the number of directives is large, but also because each directive takes multiple parameters. The parameters used with the optimizations along with the choice of optimizations often has a significant impact on the latency and area occupied by the generated RTL.

Table 3.1: Options available in AutoESL. Each option can either be specified as a configuration directive or as a pragma in source code.

set_directive_allocation	Specify instance restrictions for resource allocation
set_directive_array_map	Maps a smaller array into a larger array
set_directive_array_partition	Partitions an array into smaller arrays or individual elements
set_directive_array_reshape	Combines array partitioning with vertical array mapping
set_directive_array_stream	Use FIFOs are used instead of RAMs for arrays
set_directive_clock	Applies the named clock to the specified function
set_directive_dataflow	Specifies that dataflow optimization be performed on the functions or loops
set_directive_dependence	Allows the user to explicitly specify dependence information
set_directive_expression_balance	Rearrange expression operations to create a balanced tree
set_directive_function_instantiate	Create a unique RTL implementation for each instance of a function
set_directive_inline	Inline a function
set_directive_interface	Specify how RTL ports are created during interface synthesis
set_directive_latency	Allows a maximum and/or minimum latency value to be specified on a function
set_directive_loop_flatten	Flatten nested loops into a single loop hierarchy
set_directive_loop_merge	Merge all the loops into a single loop
set_directive_loop_tripcount	allows the user to loop trip-count
set_directive_loop_unroll	Unroll a loop either completely or by a specified factor
set_directive_occurrence	Specify that code executes at a lesser rate than enclosing function or loop
set_directive_pipeline	Pipeline a given function or loop
set_directive_protocol	Specify code region where no clock operations will be inserted by AutoESL
set_directive_resource	Use specific library resource be used to implement a variable in the RTL

3.2 Array Partitioning

Array structures in high-level design descriptions are implemented as memory blocks by default. This saves considerable amount of area than compared to using flip-flops to store individual elements, especially when the arrays are big. However, mapping arrays to available RAM resources creates resource constraints that may limit the ILP in the design. Each physical RAM block has a fixed number of read and write ports, and can therefore support only those many number of loads and stores in parallel. It is sometimes desirable to obtain a 'banked' implementation of an array, where the array elements are divided and mapped onto

```
void testAP(int a[10], int i)
{
    a[i] = i;
}
```

Figure 3.1: Example design to study array partitioning

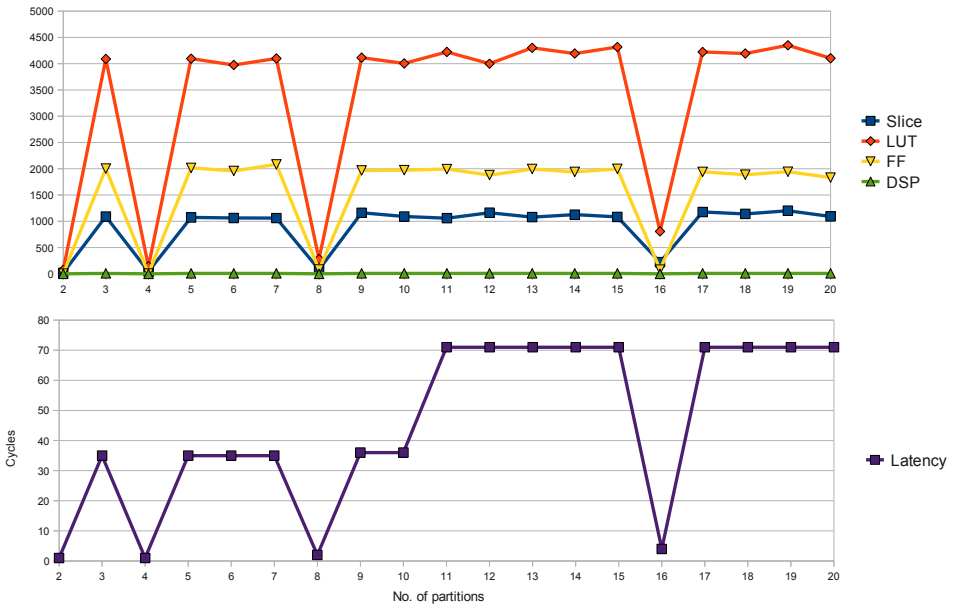

Figure 3.2: Comparison of latency and area numbers for different partitions for design in 3.2. Power-of-2 partitions perform better.

several RAM blocks. This effectively increases the number of read/write ports to the array and can alleviate the resource constraint problem. However, choosing the right splitting mechanism and number of banks is extremely crucial to obtain good benefits. A wrong partitioning choice can have a big negative impact not only by itself, but also on other optimizations. In this study, we concentrate only on cyclic distribution of array elements to different partitions.

For example, consider a simple piece of code as shown in Figure 3.2. Figure 3.2 shows the effect of various partition numbers on the latency and area. This is obviously a very simple contrived case where array partitioning is not needed, but it serves as a good example to

understand the effects of array partitioning. In particular, we make the observation that the best choices for the number of partitions seem to be powers of 2. This can be explained by observing and understanding the array partitioning optimization in action. When an array is partitioned, some additional code gets added into the design. This is required to select the right array bank and the right index within the bank. For instance, if the array a above has been divided into 3 partitions, each store into a[i] will require selecting the right array bank depending on the value of i. Hence additional code is inserted that performs i *mod* 3, based on the result of which a series of select statements will select the right bank. It turns out that the *mod* operator is very expensive as it is slow and occupies a large area. However, *mod* operator on powers-of-2 numbers are very cheap as it can be implemented by examining the appropriate number of least significant bits (LSB) and just truncating the rest. For example, to perform i *mod* 2, the least two significant bits of i provide the answer instantly. Similarly, least three significant bits are used to perform i *mod* 4, and so on. To perform *mod* on any other number, a 32-bit *mod* operator is required.

3.3 Loop Unrolling

Loop unrolling is a popular, commonly used optimization that fuses successive loop iterations into the same iteration. The number of iterations fused together is called as the *unroll factor*. Loop unrolling reduces the number of exit checks in a loop. Also, as consecutive loop iterations are merged into the same body, unrolling can potentially improve ILP. Loop unrolling also exposes more opportunities to other optimizations like scalar replacement and dead code elimination, especially in loops with a carried dependency.

in order to understand and illustrate the effects of loop unrolling from a HLS perspective, we consider two simple kernel loops shown in Figure 3.3. The first loop is a *daxpy*-style data-parallel vector computation operating on integer elements. The second loop is a sequential prefix sum implementation with a loop carried dependence across one iteration.

Figures 3.4 and 3.5 show the latency and area numbers of the loops in Figures 3.3(a) and 3.3(b) respectively. We make the following observations:

```
#define N 500
void daxpy(int a[N], int b[N], int k, int c)
{
   int i;
   L1:for(i=0;i<N;i++)
   {
      a[i] = b[i] * k + c;
   }
}
```

(a) No dependency

```
#define N 500
void prefix(int a[N+1], int b[N+1], int k, int c)
{
   int i;
   L1:for(i=1;i<N+1;i++) {
      a[i] = a[i-1] + a[i];
   }
}
```

(b) Dependence distance = 1

Figure 3.3: Two simple kernels subject to loop unrolling.

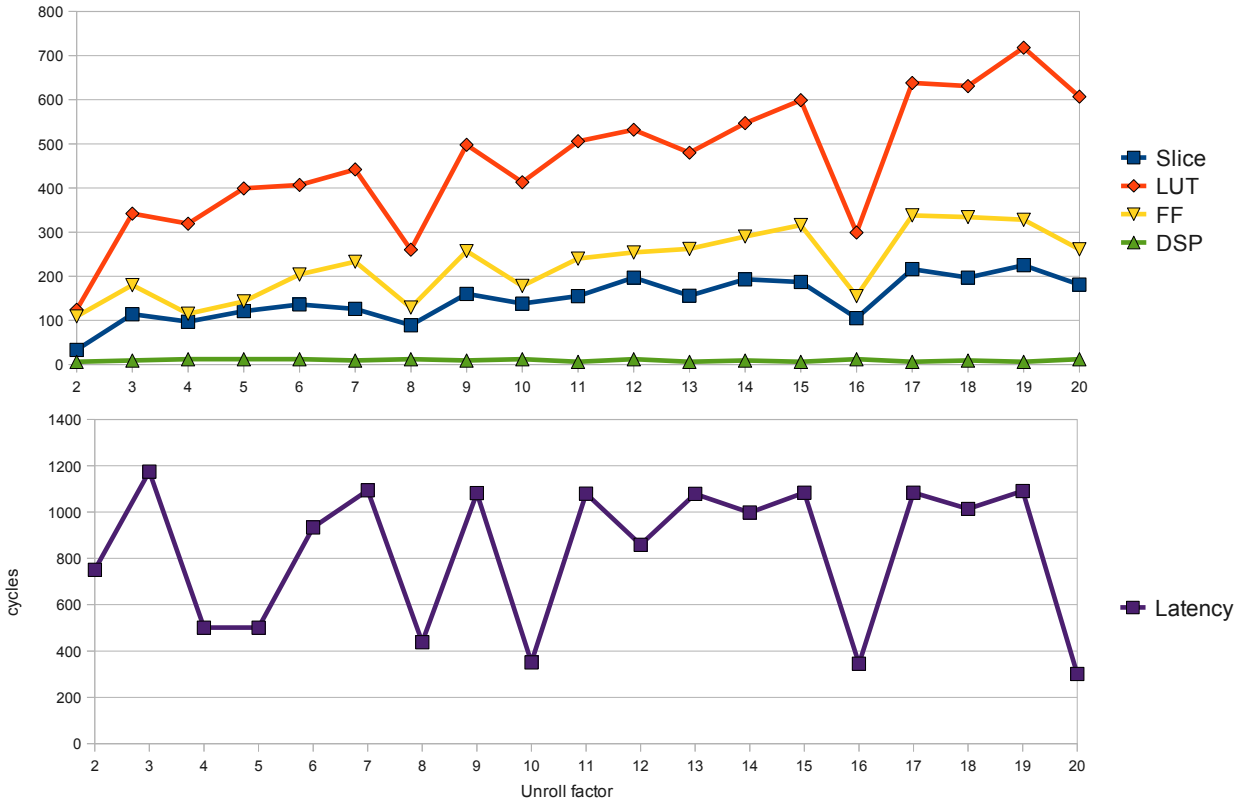

Figure 3.4: Latency and area numbers for loop in 3.3(a).

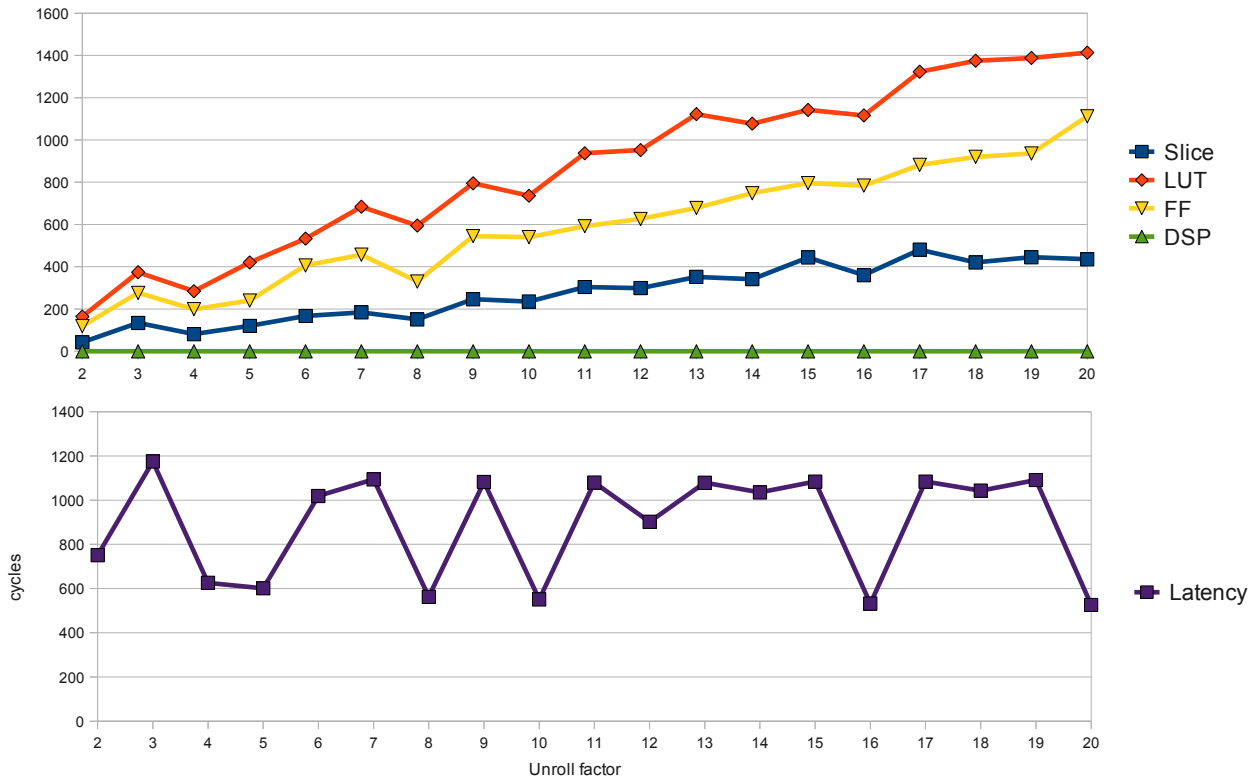

Figure 3.5: Latency and area numbers for loop in 3.3(b).

- We can observe that the best performing unroll factors in both the kernels considered are *2,4,5,8,10,16,20*. While it is not immediately obvious from these results alone, further experiments with different trip-counts revealed that the set of best unroll factors consists of both the factors of the loop trip-count as well as all powers of 2 lesser than the trip count. This can be explained by understanding why the other factors are bad choices. Unrolling a loop with a number that is not a factor of the trip-count leaves some remainder iterations to be executed. In the actual implementation, this causes the addition of a number of exit checks and branches at various stages in the loop which add considerable overhead. For non-power-of-2 unroll factors, the exit checks need a full *32-bit* comparator to check the index variable's size. Also, as the exit checks are dependent on the loop index variable, they introduce data dependency with the instructions that update the loop index variable. This limits the available ILP. Due to these reasons, the finite state machine created for this design during the scheduling phase is bigger with more nodes and much more complicated, thereby needing greater area (LUTs, multiplexers) to be implemented.

  Limiting the unroll factor search space to just factors of the trip-count will lose some opportunities. Power-of-2 unroll factors are also good choices because they simplify exit check comparisons, needing smaller (and hence faster) comparators. Considerably lesser number of exit checks are generated for power-of-2 unroll factors, thereby not adding too much overhead in each iteration. Hence, the finite state machine of the design has fewer nodes and is relatively less complicated, thereby costing lesser area as well.

- Area consumption increases linearly with unroll factor. This is an obvious conclusion because unrolling exposes many instructions that could execute in parallel. Hence the scheduler gets extra freedom to pack more instructions into each cycle, and hence there is less re-use. Also, as each iteration in the unrolled loop is much bigger, it increases the number of nodes in the finite state machine, thus increasing area.

- Unrolling dependent loops enables optimizations like scalar replacement. As loads

from memory are expensive operations, unrolling dependent loops is a good choice. However, the reduction in latency due to saving loads quickly balances out with the increase in the size of each iteration.

- In both cases discussed here, it is clear from Figures 3.4 and 3.5 that the latency gain from unrolling (using good unrolling factors) quickly flattens out. For example, the relative gains between unroll factors 8 and 10 is far lesser than the gain between factors 5 and 8.

From the observations above, we form the following conclusions:

- The set of good unroll factors S for a loop L with a trip-count of n can be defined as follows:

$$S = \{f_i | mod(f_i, n) = 0\} \bigcup \{2^k | (k \in N) \wedge (2^k \leq n)\} \tag{3.1}$$

- As area increases linearly with unroll factor, an incremental search algorithm can be used to search S for an unroll factor that provides a reasonable performance-area trade-off. The initial latency and area occupied by the completely rolled version of the loop is considered as the starting point. Starting from the lowest unroll factor s_i in S, the loop in consideration is unrolled with factor s_i. The latency and area of the loop are measured. For area numbers, we need only a rough approximate such that relative increases in area are captured 'accurately enough'. For this purpose, we use the area numbers reported by AutoESL and skip the lengthy process of logic synthesis, map and place-route. After the latency and area numbers are obtained, we calculate the *latency slope* l and *area slope* a. Based on the values of l and a, one of the following actions are taken:

  - If $|l| > |a|$ and $l < 0$, then the unroll factor s_i is good. The next unroll factor will be compared to the results of s_i.
  - If $|l| < |a|$, then the latency gain is smaller compared to the area it occupies. The algorithm stops and previous unroll factor is returned.

– If $l > 0$ and $a > 0$, then s_i is clearly a bad choice. In our current implementation, we remain optimistic by just skipping over s_i and moving on to the next unroll factor in S

The above algorithm can easily be generalized to sacrifice either latency or area. For example, in order to favour latency, we can modify the first condition by adding a weight $w > 0$ to the $|a|$ parameter. This will allow unroll factors that still achieve latency reduction, but causing a greater increase in area.

3.4 Loop Pipelining

Software pipelining is another popular loop transformation that also attempts to exploit ILP, but in a different way. Software pipelining attempts to re-order instructions such that independent instructions across iterations are executed in parallel. The goal of software pipelining is to start execution of the second (or $i + 1$th) iteration in as short an interval as possible. This interval between the launch of two successive iterations is called as the *initiation interval (II)*. Software pipelining increases design throughput. In the steady state, effectively each iteration takes II cycles to complete. Therefore, pipelining a loop with low II yields a very high throughput.

To illustrate the benefit from pipelining, consider again Figure 3.3(a). Figure 3.6 compares the latency and area of the unpipelined loop with a pipelined version having an II=1. As we can see, almost a 3X reduction in latency is achieved with a modest 20% increase in the number of slices.

A common practice is to use loop unrolling and loop pipelining in tandem. If a loop with II=1 is unrolled twice and pipelined, and if the minimum II that can be achieved with the unrolled loop can still be 1, then in the steady state each iteration in the original rolled loop takes just 0.5 cycles in the pipelined version. In general, it is profitable to unroll and pipeline a loop as long as the minimum achievable II is increasing at a slower rate than the unroll factor.

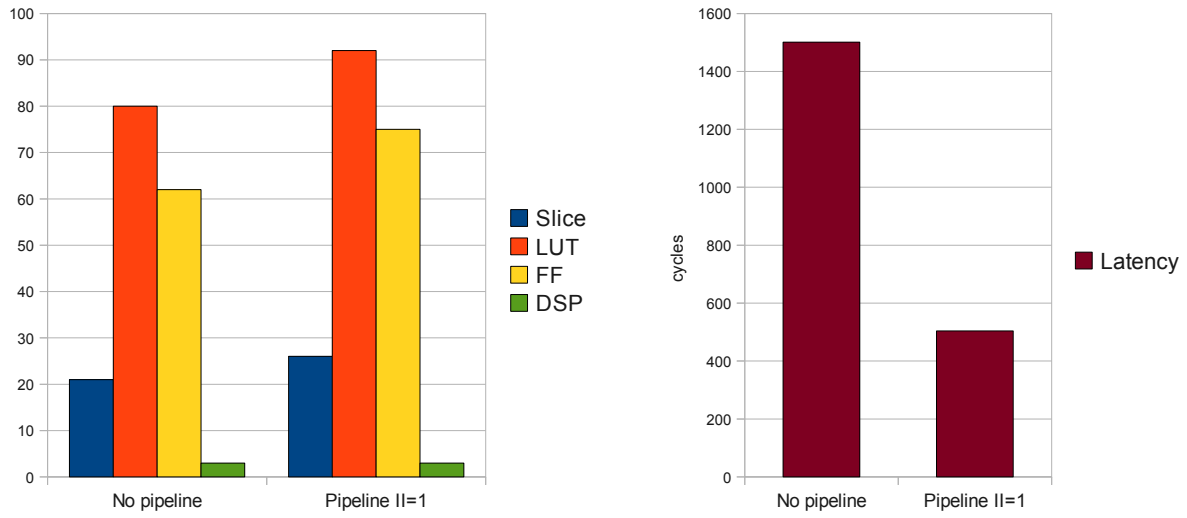

Figure 3.6: Comparison: No pipelining v/s Pipelined loop (II=1)

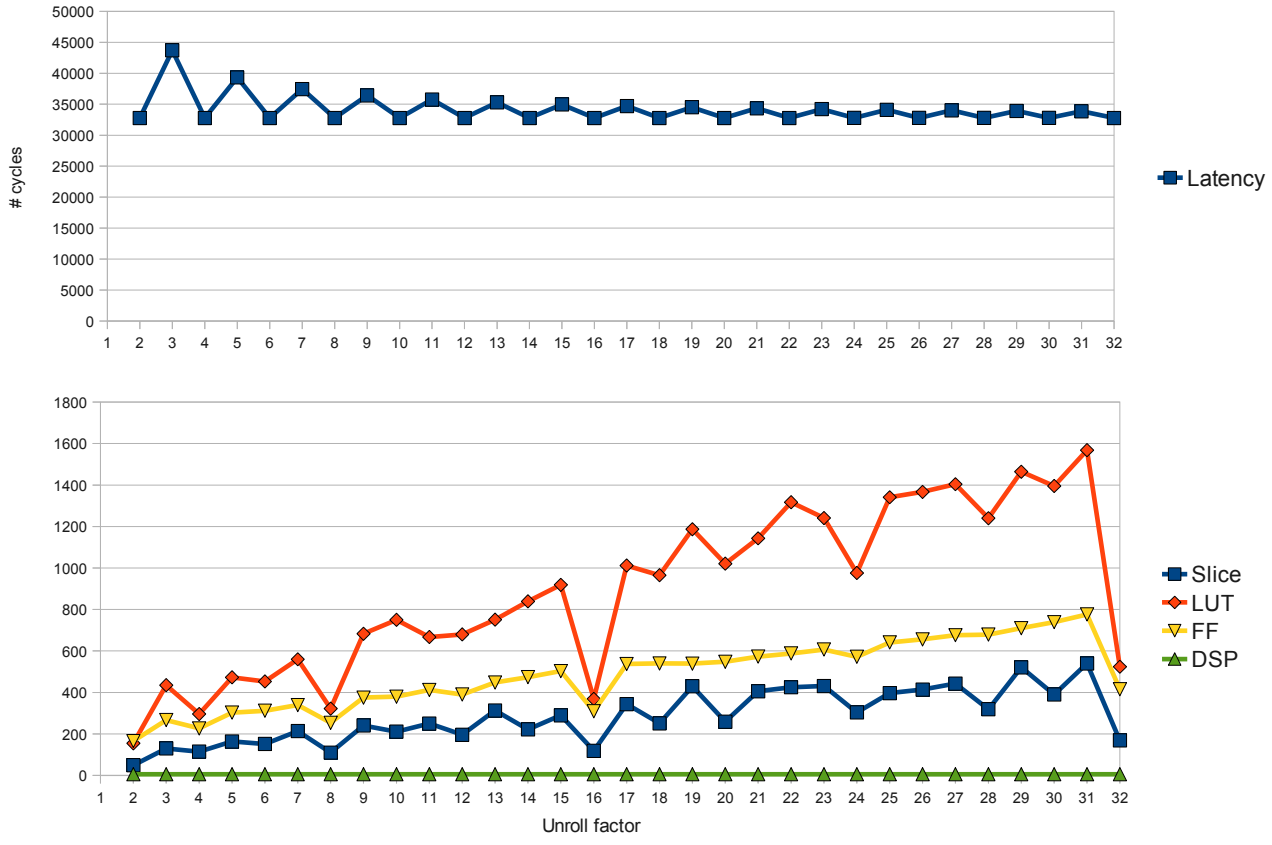

Figure 3.7: Pipelining with unrolling loop in 3.3(a) for 65536 iterations.

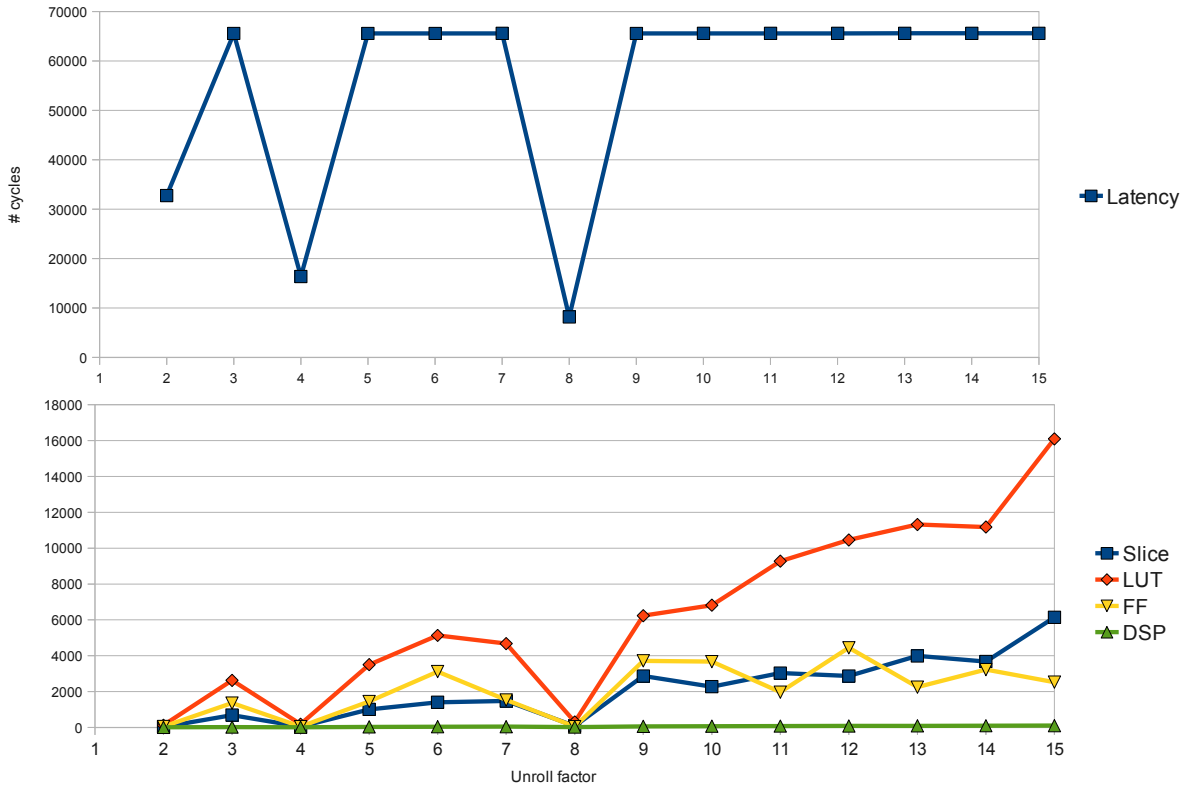

Figure 3.8: Pipelining with unrolling and partitioning 3.3(a) for 65536 iterations.

However, software pipelining can be constrained by the available memory bandwidth. Consider Figure 3.7 for instance. Here the loop in 3.3(a) has been executed for a greater number of iterations (65536) for a better illustration. While unrolling is causing a linear increase in area, software pipelining is unable to achieve the required II due to a resource constraint in terms of the number of read/write ports. With appropriate array partitioning, software pipelining combined with loop unrolling proves to be a powerful combination. Figure 3.8 shows just that. We can clearly observe gains for unroll factors of 2,4 and 8, all of which happen to be factors of the trip-count (65536) and a power of 2, making them good candidates for array partitioning.

3.5 Approach to search optimization space

We adopt the following simple approach in order to arrive at a loop configuration that achieves a reasonable performance to area. We use the algorithm described in section 3.3 to obtain the unroll factor u_i giving best performance to area. The loop is unrolled u_i number of times and then pipelined. We leverage the detailed information that AutoESL provides regarding the minimum pipelining II achieved, resource and data constraints. If the II is constrained due to memory resources, the appropriate array is subjected to partitioning. The partition factor starts at 2 and is then doubled in subsequent iterations if the previous partition factor was insufficient to resolve the resource constraint. We have found that this approach is simple, yet is very effective in quickly arriving at optimal loop configurations.

In the next section, we evaluate our approach on a set of benchmark designs.

3.6 Evaluation

In this section, we evaluate the methods and heuristics presented in the previous section. We first discuss our experimental setup and flow. We then describe the set of kernels we have chosen for evaluation and discuss results.

3.6.1 Experimental setup

Figure 3.9 shows the structure of our evaluation flow. For purposes of evaluation, we use AutoESL, an industry-standard High-level synthesis tool. We obtain area numbers from the EDA tool-chain provided by Xilinx. The target platform we consider here is Xilinx Virtex 5. However, there was no specific reason to choose Virtex5 over others, and the experiments can be run just as easily targeting any other platform. Currently, we can handle benchmarks whose loop trip-counts can be determined at compile-time.

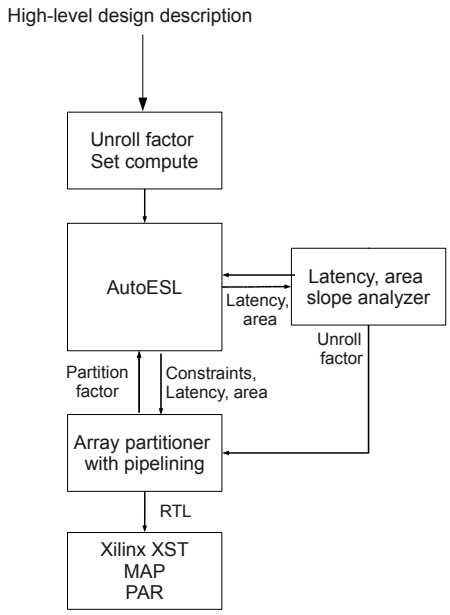

Figure 3.9: Structure of our experimental flow.

3.6.2 Results

We tested our approach on five different kinds of kernels taken from the Open Accelerator repository [1] and MiBench [15]. We have hand-chosen different kernels in order to achieve a broader evaluation coverage. The benchmarks are described below :

- adpcm_decoder: Kernel function of the ADPCM decoding algorithm. The main loop consists of a carried dependency through a scalar variable. Every iteration reads one element from an *inbuffer* and writes one element to the *outbuffer*. The trip-count of this main loop is 500.

- daxpy: A simple kernel function performing the vector operation $A = Bk + c$, where k and c are integer constants. The size of vectors A and B is 500, and the loop trip-count is 500.

- prefix: A kernel function calculating prefix sum on a vector of integers. The size of the array and the loop trip-counts are both 500. The loop carries a dependency through the vector.

- segmentation: Kernel performing one compute step in an image segmentation algorithm. The kernel consists of a nested loop with lots of floating point computations and *sqrt* operations. This loop is data-parallel. We currently handle the innermost loop in the nest. AutoESL automatically flattens the loop nest if any outer loop is attempted to be pipelined. We do not explore that here currently. The inner loop runs 64 times.

- smithwaterman: An implementation of the Smith-Waterman algorithm. This algorithm also contains nested loops with the core innermost loop having a trip-count of 76. This implementation contains many two-dimensional arrays and the loop body performs considerably many array loads and stores, more than any other benchmark that we have considered here. The loop carries a data dependency through an array.

Table 3.2: Comparison between baseline and heuristically optimized benchmark versions against latency and area using ER. The unroll and partition factors used are also specified.

Benchmark	**Unroll factor**	**No. of partitions**	**Numbers**							
				Slice	LUT	FF	II	Depth	Latency	ER
adpcm_decoder	4	1	Baseline	200	588	217	-	-	2502	1
		1	Baseline + PP	224	741	234	2	5	1006	2.22
		1	U4 + PP	619	2009	471	8	11	1006	0.8035
daxpy	8	1	Baseline	21	80	62	-	-	1501	1
		1	Baseline + PP	26	92	75	1	3	504	2.405
		4	U8 + PP	89	324	315	1	3	67	5.286
prefix	8	1	Baseline	29	113	80	-	-	1501	1
		1	Baseline + PP	43	166	91	2	3	1003	1.009
		8	U8 + PP	109	307	375	2	4	130	3.072
segmentation	32	1	Baseline	31	110	65	-	-	8321	1
		1	Baseline + PP	43	153	88	1	2	4100	1.463
		16	U32 + PP	173	522	160	1	3	132	11.296
smithwaterman	4	1	Baseline	26	102	46	-	-	52281	1
		1	Baseline + PP	19	73	46	2	3	11708	6.110
		1	U4 + PP	-	-	-	-	-	-	-

Table 3.2 shows the optimal factor obtained, number of partitions required, latency and area numbers for all benchmarks. Each benchmark is reported under three configurations:

Baseline - where the benchmark was run without any high-level optimization, *Baseline + PP* - baseline with pipelining, where the main loop was pipelined with the required number of array partitions, and $U(num) + PP$ - Unroll by obtained unroll factor with pipelining along with required number of array partitions.

We define the *efficiency ratio* ER as the latency-area product, as follows:

$$ER = \frac{latency_b * area_b}{latency * area} \tag{3.2}$$

Here, $latency_b$ and $area_b$ are the latency and area numbers of the baseline respectively. We use the number of slices occupied as the representative for area of a design.

The ER is basically a ratio of the speedup v/s area increase, and gives an indication about the how well the area has been utilized. We can see the ER increases with increase in speedup and decrease in area utilization. The higher the ER, better is the result.

We make the following observations:

- *adpcm_decoder* does not benefit from unrolling. In fact, unrolling degrades its performance as can be seen by its ER. This can be explained by examining the structure of the loop. As mentioned before, the *adpcm_decoder* loop contains a dependency across iterations carried over by a scalar. Due to the carried constraint, unrolling the loop just concatenated more instructions from the next iteration to the end of the loop body. As a result, the II scaled linearly with increase in the unroll factor, thereby mitigating the effects of software pipelining. Hence the best result for this benchmark is when unrolling is at its minimum i.e., the loop is completely rolled. Also, as the constraint is a data dependency, array partitioning cannot help alleviate the constraint.

- Benchmarks *daxpy* and *prefix* perform well, with ERs of 5.2 and 3 respectively. It is to be noted that the achieved performance is due to the combined effect of array partitioning, loop unrolling and software pipelining. Further experiments showed a decrease in performance if either one of these parameters were altered. *Prefix* has a slightly smaller gain than *daxpy* due to the carried constraint in the loop.

- *Segmentation* achieves a remarkable benefit with its configuration with an ER of around 11. As the core loop is data parallel, the only constraint to achieve minimal II would be array resources, and a partitioning factor of 16 resolves all resource constraints.

- *Smithwaterman*'s numbers are surprising. The area occupied by the pipelined baseline version is lesser than the unpipelined version. This suggests that the schedule is utilizing the hardware very efficiently. Also, SmithWaterman benefits from pipelining with an ER of 6. Using our heuristics, an unroll factor of 4 was found to give the best performance to area value. However, pipelining the unrolled loop resulted in an AutoESL crash due to an internal bug in the tool.

Although some benchmarks we have chosen to evaluate our approach are relatively small in size, they are kernels which represent real-world computation. We see that the right combination of partition factor, unroll factor and pipelining can improve the performance by an order of magnitude.

CHAPTER 4

IR-level optimizations

In this chapter, we describe our study on the effects of phase-ordering of IR-level optimizations. Here, we use the latency of the RTL as the metric representing the quality of the RTL. We first describe the various optimizations used and how we chose them. We then describe several techniques that we have implemented and evaluated. We also compile and produce binaries using the same sequence used in HLS, simulate them on a cycle-accurate out-of-order processor simulator and compare the ranks of each sequence on a processor and in HLS.

4.1 Motivation

We motivated our study by discussing a simple example in Chapter 1. In this section we study the same example along with another simple design pattern in the HLS perspective only, to highlight the significance of compiler optimizations on the generated RTL.

Consider the two example programs shown in Figures 4.1 and 4.2. We consider the same set of optimizations that we consider in Chapter 1.

- -gvn (g): Global value numbering.
- -mem2reg (m): Promote memory to register.
- -indvars (i): Canonicalize induction variables.

Figure 4.3 provides the performance of three combinations - *img*, *mig* and *gim* on the two benchmarks. Two things are evident in this example:

```
void add(int i[20], int o[2])
{
    int iter;
    o[0] = o[1] = 0;
    for(iter = 1; iter <= 20; iter++)
    {
      if(i[iter]%2 == 0) {
         o[0] += i[iter];
         o[1] += i[iter];
      }
    }
}
```

Figure 4.1: Motivational example - simple addition.

```
void fact(int *i, int *o)
{
   int n = *i;
   int fact = 1;
   int iter;
   for(iter = 1; iter <= n; iter++) {
        fact = fact*iter;
   }
   *o = fact;
}
```

Figure 4.2: Motivational example - factorial.

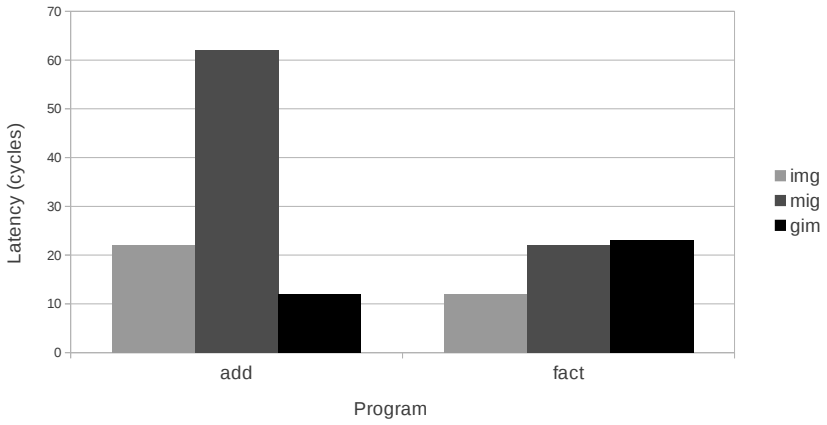

Figure 4.3: Latency comparison for different optimization orders for examples in Figure 4.1 and Figure 4.2.

- There is a noticeable variance in the latencies of the RTLs generated with different optimization combinations. The latency of *add* varies from 12 to 62 while the latency of *fact* varies from 12 to 23
- The *best* sequence is different for the two programs. In case of *add*, it is sequence *gim* whereas in case of *fact*, it is *img*.

Although the example considered above is small, it helps illustrate the effect that the order of application of transformations has on the resulting RTL's latency.

As a second case study, we examined a benchmark *matrixmul* that performs tiled matrix multiplication. This benchmark is greater in terms of both size and complexity than the example above. We considered 68 different optimizations and applied several transformation sequences on *matrixmul*, where each transformation sequence was of length 200. All optimizations were applied using xPilot's LLVM-based HLS front-end. The transformed output from xPilot was passed through AutoESL, a commercial High-level synthesis tool. We further synthesized the generated RTLs using XST, Xilinx's logic synthesis tool in order to get accurate area estimates.

Figure 4.4 shows the normalized latency and area numbers for a few sequences. We can

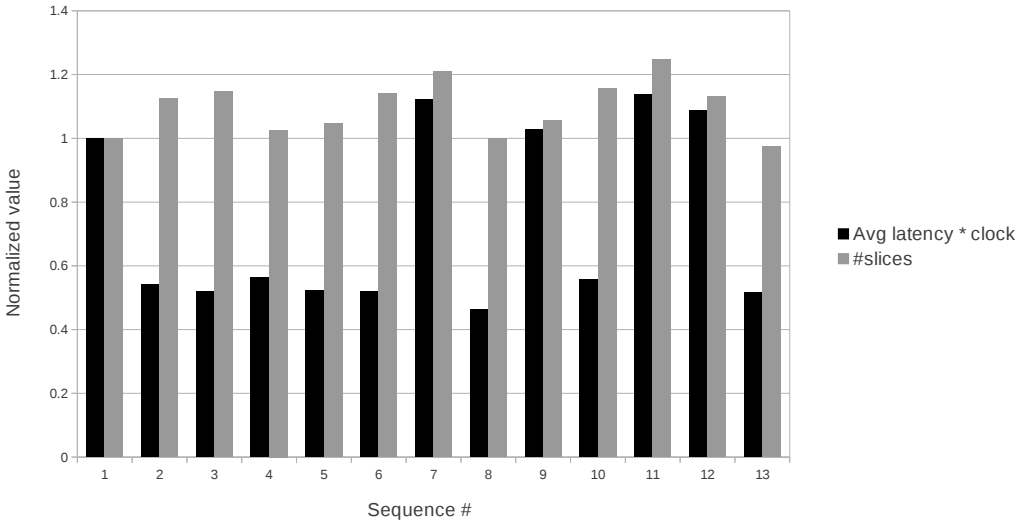

Figure 4.4: Normalized latency and area numbers for *matrixmul* obtained from AutoESL and Xilinx back-end.

see that a reduction in latency of almost 2X can be obtained, with a very modest increase in area. This shows that applications can derive significant benefit with the correct optimization order.

Another aspect that can be noticed is the irregularity and discreteness of the space of optimizations. As can be seen in later sections, latency does not take on a spread of values for different optimization sequences; there are discrete clusters of good and bad sequences. Such a *step-constant* nature of the optimization sequence space is not a new finding. Previous research [18][4] has shown the same result. This result suggests that the latency is affected only by a few "critical" optimizations. Section 4.2 provides further details on our investigation of this property.

We used AutoESL synthesis tool only as part of our initial study to explore the possible gains that could be achieved. While the encouraging results from such a tool provides a strong motivation, most commercial tools are licenced and are not open source. Also, it is likely that the tool performs many transformations on the input which cannot be turned off, and hence it would not be possible to correctly evaluate the sequences that we use to

transform the input before passing it through the tool. On the other hand, xPilot is an academic tool which provides us with maximum flexibility to accurately test the effect of phase orders. Therefore, for all our experiments mentioned henceforth, we use only xPilot.

4.2 Methodology

In this section, we first describe the various optimizations used and how we chose them. We then describe several techniques that we have implemented and evaluated to search the optimization search space. We introduce the simple idea of *lookAhead* in Section 4.2.4. We then describe a technique that we tried to speed up our iterative algorithm using a static latency estimation model.

4.2.1 Optimizations considered

In the default setting, xPilot applies close to 250 transformations from a set of 68 unique optimizations. While most of the optimizations in this set are standard compiler transformations like global value numbering, strength reduction etc., a few optimizations are specific to High-level synthesis. One example for such an optimization is the *bitwidth minimization* optimization. Unlike a conventional CPU, reconfigurable fabric like FPGAs can support arbitrary data sizes. This optimization can be very beneficial, as shown section 4.3. Intuitively, say we have an integer multiply instruction $a*b$ in the input program. If the maximum range of values taken by a and b can be statically estimated to not exceed 256, an 8-bit multiplier can be used to perform this operation as opposed to a 32-bit adder, thereby conserving area and also decreasing latency.

As mentioned earlier, the space of optimization sequences is discrete. To investigate this, we generated 1000 random sequences of length 200, where optimizations were taken from the same set that is used by Xpilot. While this is by no means exhaustive, it is sufficient for understanding some aspects of the solution space. Figure 4.5 shows the scatterplot of the average latencies obtained from the sequences. One can immediately observe the discrete horizontal steps and blank spaces in-between. We repeated the same experiment on multiple

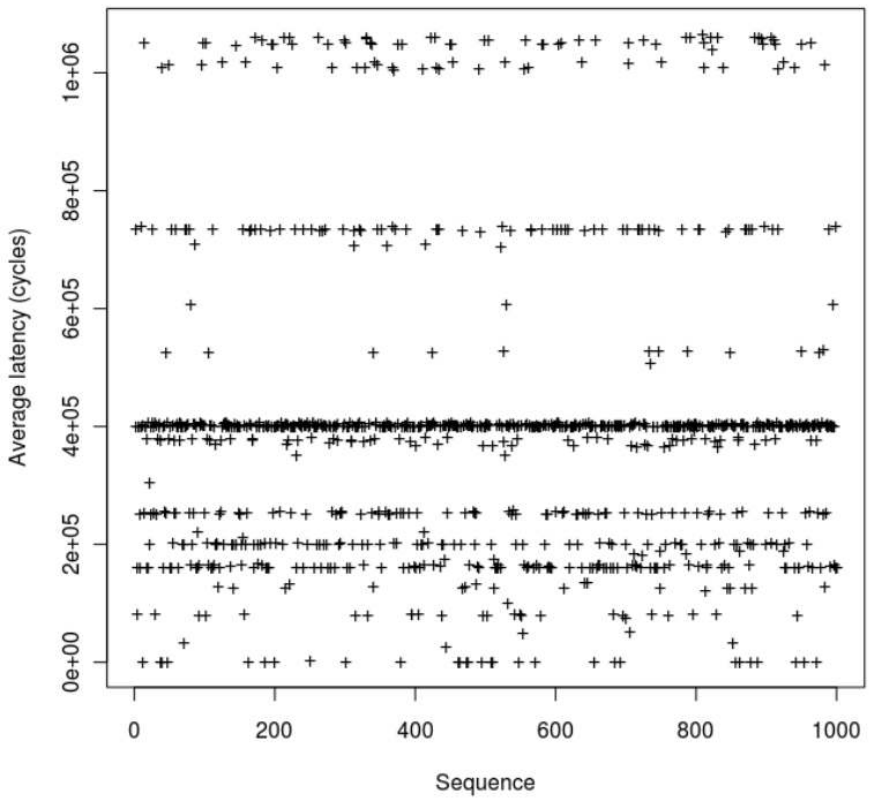

Figure 4.5: Scatterplot of latencies for *matrixmul*

benchmarks, only to see a similar result.

Intuitively, such behavior is caused when a few optimizations are far more important than the others. In order to empirically support our claim, we randomly chose 100 sequences and examined the effect of each optimization on latency in a step-wise fashion, with the hope of finding the critical jump points. In other words, if the sequence under consideration is *abcde*, we examined the change in latency due to *a* alone, then *ab*, then *abc* and so on. Table 4.1 shows the result of one such examination.

Table 4.1 clearly shows the critical points where there is an observable change in latency. Similar experiments with other sequences and other benchmarks yield similar results with discrete changes. Although the order of optimizations was different in each case, there was a large overlap among the set of optimizations occurring in such critical points. We performed this exercise on 100 randomly chosen samples from five different benchmarks, and listed

Table 4.1: Step-wise analysis of one sequence. Reported latency is reported (in cycles) from xPilot. We used a target clock period of 50 ns.

Position	Optimization	Latency change (cycles)	Final (cycles)
2	instcombine	−27000	708094
22	inst-simplify	+27000	735094
40	loop-deletion	+3520	738614
42	indvars	−6682	731932
66	gcse	+316536	1048468
68	gvn	−647792	400676
74	simplifycfg	−400627	49

the most commonly occurring optimizations. Table 4.2 gives a brief description of all the short-listed optimizations. From here on in this paper, we restrict all our experiments to this restricted subset of optimizations.

4.2.2 Random search

In our implementation of random search, we generate random sequences containing upto 25 optimizations each. We allow repetitions of optimizations, since it resembles a more natural setting. We generated and evaluated 5000 random sequences for each of the benchmarks considered. Section 4.3 has further information on our evaluation.

4.2.3 Genetic Algorithm

We implement a genetic algorithm to search the space of optimization sequences using latency as the minimization cost function. Genetic algorithms are iterative algorithms that converge to a solution mimicking the process of natural evolution. In our implementation, we chose to have a randomly generated initial population of 20 sequences, each of which can have as many as 25 optimizations. We repeat the iterative search process for 500 generations. In each iteration, all the sequences in the population are evaluated and ranked. The sequence

Table 4.2: Subset of optimizations and descriptions.

name	description
adce (a)	aggressive dead code elimination
bitwidthmin (b)	bitwidth minimization
condprop (p)	conditional propagation
constprop (k)	constant propagation
dse (e)	dead store elimination
gcse (c)	global common subexpression elimination
gvn (n)	global value Numbering
indvars (v)	canonicalize induction variables
instcombine (i)	combine redundant instructions
inst-simplify (t)	operator strength reduction
loop-deletion (d)	delete dead loops
loop-preproc (o)	Loop preprocess
loop-simplify (l)	canonicalize natural loops
mem2reg (m)	promote memory to register
ptr-legalization (r)	convert pointers to array indices
simplifycfg (s)	simplify the control-flow graph
xunroll (x)	partially unroll loops

with the lowest latency among all is given rank 1, while the sequence with greatest latency is given rank 20. At the end of evaluation, sequences in the population undergo mutation and crossover as follows. Four bottom-most sequences having the greatest ranks are removed. They are replaced with new sequences formed by crossing over the top four sequences. To perform a crossover, we concatenate the first half of one sequence with the second half of another sequence. The best sequence is carried through to the next generation unaltered. As a last step, 8 to 10 sequences are randomly chosen and mutated. Our implementation chooses sequences at the bottom with a higher probability to be mutated. To perform a mutation, we randomly change an arbitrary number of flags in the sequence. Finally,

duplicate sequences are replaced with random sequences. The best solution found after 500 generations is reported.

4.2.4 n-lookahead scheme

One way to construct an optimization scheme is to progressively consider the best alternative among all choices and append the best choice to the existing sequence. The n-lookahead scheme is based on an observation that the number of optimizations enabled or disabled by each optimization is relatively small. A bad sequence typically consists of a few misplaced optimizations which cause a cascade of disabling interactions. In other words, we can think of an optimization sequence as a sequence of smaller subsequences, where only the subsequences need to be thoroughly examined for interactions. If the length of such a subsequence is n, constructing an optimization sequence involves selecting the best set of n-length subsequences. In this way, we are effectively *looking ahead* by n steps and choosing the subsequence that gives the best overall benefit at each step.

A 0-lookahead scheme is basically a greedy approach where we evaluate and choose the best optimization at every stage during the construction of a sequence. Likewise, a n-lookAhead scheme where n is the length of the sequence itself is nothing but an exhaustive search algorithm employed on the whole sequence. From a perspective of capturing all interaction behaviour, we would want n to be as large as possible, thereby having a greater window of lookAhead into the possible enabling and disabling effects. However, increasing n to a large number will exponentially increase the number of combinations to be evaluated. If we have to construct a sequence of length k with n levels of look ahead, and we have N number of unique optimizations, the number of combinations to be evaluated is $(k/n) * N^n$. The value of n should therefore be small enough to have a reasonable number of sequences to evaluate, while large enough to capture the essential interactions successfully.

In our current implementation of lookahead-0 and lookahead-1, we exhaustively examine all possible choices at each stage of operation before making a decision. That is, we examine all N optimizations for lookahead-0 and all N^2 pairs of optimizations for lookahead-1 at each

stage. We then sort and rank all choices based on the benefits obtained, where the sequence achieving most reduction in latency is given rank 1. In case there is a tie between two candidate subsequences that give the same benefit, we break the tie by arbitrarily choosing one of the sequences in alphabetical order. A special case is when none of the sequences offer any benefit i.e., all choices are tied at 0 benefit. In such a case, we try all the choices to see if some other optimizations might get enabled after we apply them. In case no optimization was found to be enabled, the search stops and reports the best sequence and latency found. In section 4.3, we evaluate the effectiveness of 0-lookahead and 1-lookahead schemes.

4.2.5 Latency estimation model

We were motivated by the following reasons to build a static cost model based on input program features:

- A static cost model is much faster in providing a latency estimate than xPilot's backend. This can help significantly speed up all algorithms discussed in this work, especially for complicated designs like *smithwaterman* which take around 80 to 90 seconds to evaluate one sequence.

- A cost model based solely on input program features provides a convenient and unified way to study the interaction of transformations on the cost function at a greater detail. In other words, we will understand the behavior of each optimization in terms of the cost-affecting program features that it modifies as well as the effect that it has on the cost function.

Our initial experiments clearly showed that straight-forward features like instruction count, basic block count etc., poorly correlated with latency. In this work we attempted to find program metrics that affect RTL latency. As an initial study we have considered three simple latency estimators based on three program metrics. The metrics are described below:

- The longest weighted data-dependent path: In order to extract this metric, we first build a data dependency graph (DDG) using the *def-use* chains in the code. Each node

in the DDG corresponds to one instruction. Our data dependency graph is acyclic, in the sense that we do not add back-edges to the graph. However, we annotate each node with a special *weight* W. We use the W to factor in effects such as functional units with a greater latency, instruction within loops etc. W is calculated as follows. All nodes by default have a weight of 1. Instructions like *mul* and *div* have a weight of 2. Instructions with a loop nest level of l are assigned a final weight of $W * 2^l$. The estimated latency is the sum of weights of all nodes along the longest path in the DDG.

- Longest un-weighted data-dependent path: Extraction of this metric is very similar to the metric above, except that we simply count the total number of nodes on the longest path and ignore their weights.

- Topological nodes removed per iteration: A topological sort of the DDG gives a schedule of instructions that preserves all data dependencies. When performing a topological sort on a DDG, the number of nodes that have no predecessors in each iteration is basically the number of independent instructions that are ready to be scheduled. Therefore, examining the number of nodes removed per iteration of topological sort gives a rough idea of the structure of the input code. A well known fact is that a broad, balanced-tree structure in a DDG has a shorter critical path. Such a tree-like structure would have a large number of nodes removed during the first few iterations of the topological sort, followed by continually diminshing quantities of nodes in successive iterations, which means that this parameter would have a large variance but small median. We ran experiments using the median of the number of nodes removed per iteration and discuss results in section 4.3.

Latency is influenced by a combination of the factors above and many more factors such as resource constraints, number of loops and so on. Building a good latency model is a complicated task. As we show in section 4.3, although the simple experimental models above provides reasonable performance for a few benchmarks, more number of metrics need to be taken into consideration to ensure consistency of the estimator across benchmarks.

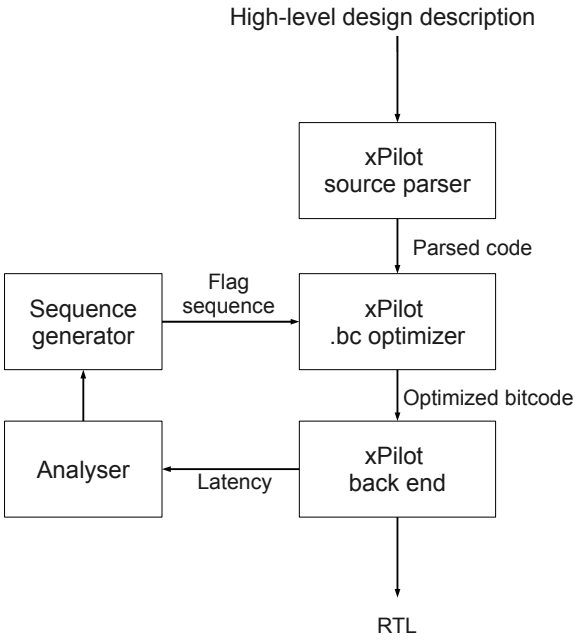

Figure 4.6: Modified xPilot structure.

4.3 Evaluation

In this section, we evaluate our approaches described in section 4.2.

4.3.1 Experimental setup

We use xPilot [5], a High-level synthesis tool for research. We slightly re-structured xPilot's flow as shown in Figure 4.6 to facilitate easier evaluation of sequences. All tests were run on an Intel(R) Xeon(R) CE5405 CPU, which has 8 cores running on a 2.00GHz clock. We have chosen the HLS target platform to be Xilinx Virtex4. There is no specific reason for choosing Virtex4; the same study could be repeated easily on targeting different platform. All tests were controlled and automated using a few python scripts. The data dependency graph construction and latency estimation were done by a pass written in LLVM.

Table 4.3 lists and describes the set of benchmarks that we consider in our evaluation process. We consider a mixture of simple benchmark designs like *binarysearch* and *honda* to complicated benchmarks like *fft* and *smithwaterman*. Another aspect to be kept in mind

Table 4.3: Benchmarks and description.

Benchmark	Description
binarysearch	Iterative binary search
cftmdl	Kernel region in 1D FFT computation
chem	DSP algorithm in a chemical plant
dir	Direct implementation of 1D DCT
fft	Fast fourier tranform from MiBench [15]
honda	DSP filter application
jacobi	Jacobi method to solve linear equations
lee	Lee's algorithm for 1D DCT [20]
matrixmul	Tiled matrix multiplication
sha	SHA-1 encryption algorithm
smithwaterman	Smith-Waterman algorithm

is that all benchmarks are behavioural descriptions that are to be implemented in hardware. Therefore, benchmarks like *sha* are fairly complicated cases. In our current study, all benchmarks contain just one function. We slightly modified a few benchmarks by manually in-lining a few helper functions so that the benchmark was contained in a single function. Extending our approach to multiple functions is relatively easy.

4.3.2 Random sampling vs. xPilot

Figure 4.7 shows the comparison between the results of random search and the default optimization setting in xPilot. It can be seen that there are significant gains that can be achieved with an optimization order that is benchmark-specific.

Benchmarks *binarysearch*, *cftmdl*, *chem*, *dir*, *honda* and *lee* are all small benchmarks. In spite of their simplicity, it can be seen that they benefit from the choice of optimizations. Upon inspecting one of the benchmarks, *honda* closely, we were able to pin-point the reason that the default setting was performing badly. The default setting applies the algebraic re-association operation (*-reassociate*) which basically re-orders expressions such that more

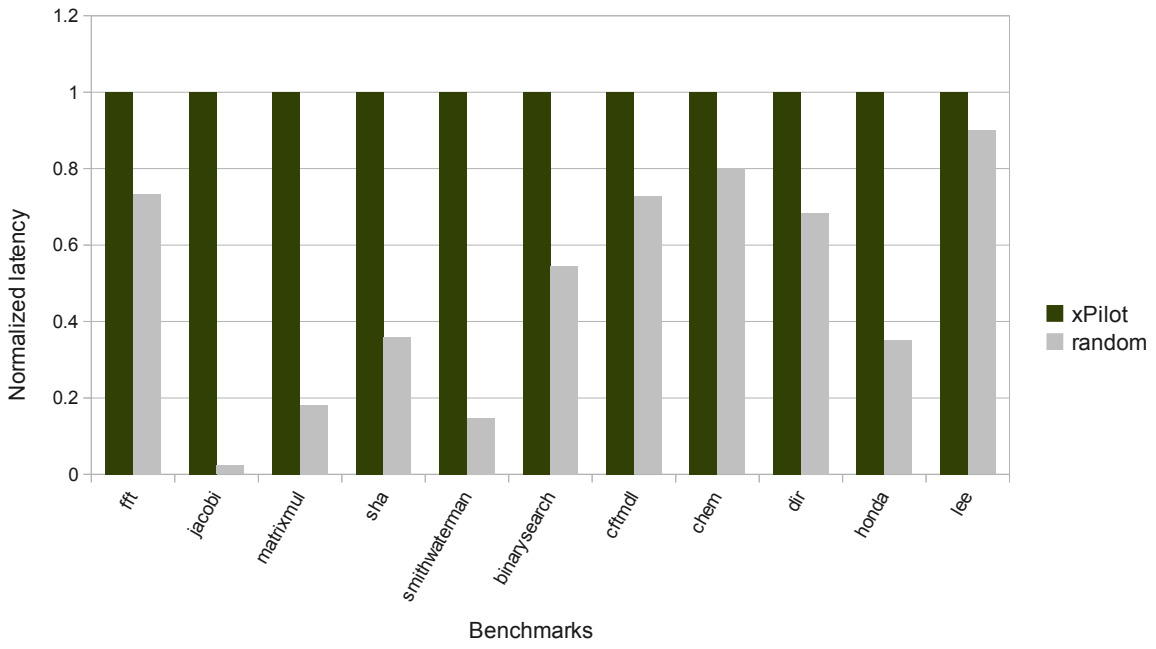

Figure 4.7: Comparison of latencies: xPilot default sequence v Random search. Results are normalized to latency obtained by xPilot's default setting.

opportunities are exposed to other optimizations following it, such as constant folding, partial redundancy elimination etc. For example, consider an expression $y = 4 + x + 5$, which would translate to a 3-address IR of $t1 = 4 + x; y = t1 + 5;$. The algebraic re-association optimization aims to re-associate such potential candidates so that the IR looks like $t1 = 4 + 5; y = t2 + x$, so that constant folding and propagation can be performed to eliminate the first instruction altogether. However, it turns out that while the re-association operation exposes a few more redundant expressions, it has a side-effect of lengthening the critical chain. We could explain (and verify by hand) that in this case, re-association was adding more skew to the data-dependency graph, and hence increasing the latency of *honda*. As our optimization subset does not contain re-association, none of the sequences generated and searched by our algorithms faced this problem.

Benchmark *jacobi* achieves a rather remarkable reduction in latency. The original latency was 20788 cycles, while the best latency searched was 492. We discuss this in greater detail later in this section.

Overall, we achieve a mean reduction in latency of 50.42% over xPilot's default setting.

4.3.3 Comparison of approaches

In this section, we compare the performances of various approaches discussed in Section 4.2.

As mentioned previously, we consider several small benchmarks in our benchmark set. While they help illustrate the fact that even small designs can benefit from good phase orders, they provide little information about the effectiveness of our approaches. All the small benchmarks achieved the exact same gain as random search from all our other approaches. The reason for this is that such benchmarks contain opportunities for only a few optimizations, and hence the interacting subset is fairly small. We discussed the nature of one such interaction with *honda* in the previous section. To better quantize the merits/de-merits of our approaches, we include only one small benchmark (*binarysearch*) as a representative example for all the other smaller benchmarks in all our further analyses.

Table 4.4 shows the comparison (in clock cycles) of the best latencies along with the best sequences obtained using different methods. Random search and genetic algorithm match up to each other in most cases except *fft*, where the genetic algorithm outperforms random search.

We can also see that a similar trend exists between random search and 1-lookahead. We consider this to be a promising result; by looking at just pairs of optimizations, we can arrive at the same solution that we could arrive at using genetic algorithms and random search. However, we can see that *matrixmul* does not match up like the other benchmarks. It is likely that a 2-lookahead scheme would have covered a few of the interactions that were missed by the 1-lookahead scheme, but we did not explore that as part of this work.

Table 4.5 shows the number of generations that the genetic algorithm took to converge to the solution reported in Table 4.4. We can immediately see that the algorithm converges pretty quickly. *fft* takes the longest time of 186 generations. Figure 4.8 shows the rate of convergence. One can observe that all the graphs are step functions containing one or two main steep drops. This strongly indicates the presence of a handful of *critical* optimizations

Table 4.4: Comparison of different approaches. Some smaller benchmarks were left out for brevity, as they closely resembled *binarysearch*.

Benchmark	Approach		Benchmark	Approach	
fft	Random	Latency: 2361 Sequence: srnaln	jacobi	Random	Latency: 492 Sequence: cbaemkxemsbsdtntsem
	GA	Latency: 1896 Sequence: mxvbvenr		GA	Latency: 492 Sequence: bmciensi
	0-lookAhead	Latency: 2359 Sequence: nktadnxpslbcoemns		0-lookAhead	Latency: 48358 Sequence: mxbtpsealodk
	1-lookAhead	Latency: 2339 Sequence: mnriksbtvbosknkacr		1-lookAhead	Latency: 492 Sequence: kmnsossbcsebnaiinr
matrixmul	Random	Latency: 49 Sequence: aooeakbesmdsttvocaosa	sha	Random	Latency: 1442 Sequence: mxxasdotxrlxlbrmnt
	GA	Latency: 49 Sequence: mceosi		GA	Latency: 1442 Sequence: iiiiiiim
	0-lookAhead	Latency: 80119 Sequence: rnevxbntoldamssss		0-lookAhead	Latency: 1442 Sequence: mxnapseixrnacsmo
	1-lookAhead	Latency: 669 Sequence: rpnevxbpdeboomisiceb		1-lookAhead	Latency: 1442 Sequence: kmsipekrbrrpemoker
smithwaterman	Random	Latency: 23 Sequence: kbvdsbbtmeosmn	binarysearch	Random	Latency: 12 Sequence: etaneb
	GA	Latency: 23 Sequence: inemsobs		GA	Latency: 12 Sequence: iiiiiiin
	0-lookAhead	Latency: 844 Sequence: kneimmissnirnxxxp		0-lookAhead	Latency: 12 Sequence: mtsnprciebvsdsak
	1-lookAhead	Latency: 23 Sequence: neamiaosaaaa		1-lookAhead	Latency: 12 Sequence: kmpkrsnciienaebmts

that account for maximum reduction in latency.

Further analysis on the best sequences for each benchmark revealed interesting results. The following are a few of the observations that we make:

- Benchmark *jacobi* achieves a remarkable reduction in latency. In order to ascertain the reason for such a drastic difference, we examined the differences between the default sequence and the best sequence. We found that the *jacobi* benchmark has subsequence *ns (-gvn -simplifycfg)* as one of its critical optimizations. We observed that the default sequence applied *-scalarrepl* optimization (scalar replacement) before *-gvn* or *simplify-*

Table 4.5: Convergence of the genetic algorithm. Some smaller benchmarks were left out for brevity, as they closely resembled *binarysearch*.

Benchmark	Generations
fft	186
jacobi	33
matrixmul	1
sha	1
smithwaterman	51
binarysearch	1

cfg. In this case, *-scalarrepl* had a disabling interaction on ns. We re-ran the default sequence by disabling just the *-scalarrepl* optimization, and obtained the same reduced latency number as shown in Table 4.4. We can also observe the presence of either ns or bs in all the good sequences for *jacobi* reported in Table 4.4. Note that the critical pair need not be applied consecutively as long as there is no disabling optimization (like *-scalarrepl* in this case) applied in between them.

- Benchmark *smithwaterman* benefits from the existence of the pair ms (i.e., *-mem2reg -simplifycfg*). We obtained this information by observing all the sequences in successive generations of sequences evaluated as part of the genetic algorithm. We noticed that a vast majority of bad sequences did not contain either m or s or both.

- Genetic algorithm search for benchmark *sha* produced an interesting sequence as reported in Table 4.4. It turns out that optimization m (i.e., *-mem2reg*) is a critical optimization. We can observe in Table 4.4 that all the reported good sequences apply m quite early in the sequence, and that i (*-instcombine*) does not affect m's opportunities. However, in order to examine the set of optimizations that do have a negative effect on m, we studied a set of bad sequences obtained from the genetic algorithm and random search. We observed that a vast majority of the sequences that had optimizations r (*-ptrLegalisation*) and t (*inst-simplify*) applied before m performed poorly as

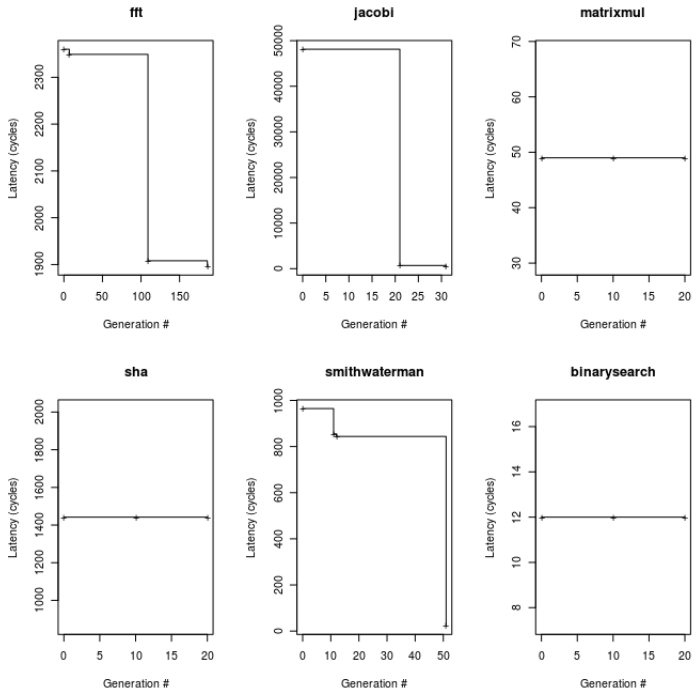

Figure 4.8: Convergence rate of genetic algorithm for benchmarks.

they negatively interacted with m.

- Benchmark *binarysearch* is far less complex and contains only a few optimization opportunities. Sequences from lookahead and genetic algorithms revealed that optimizations m or n can both optimize *binarysearch* to achieve the latency reported in Table 4.4.

4.3.4 Evaluating the latency estimation model

In this section, we evaluate the three metrics introduced earlier in section 4.2.5. To evaluate the metrics, we use them to steer the genetic algorithm described above. The initial population is initialized randomly as before, but the evaluation of sequences is done solely based on the metrics. We evaluate all three metrics in isolation by measuring the final latency that the genetic algorithm converges to after 100 generations.

Table 4.6 shows the performance of the three metrics. We can see that *sha* and *fft* perform well on the topological count metric. The weighted path performs poorly on all three benchmarks, suggesting that the weight scheme used is not modeling latencies correctly.

Table 4.6: Evaluation of latency estimation metrics. The metric in the first column was used to guide the GA, and a best sequence was obtained. The numbers reported are the final latencies from xPilot upon applying the best sequence obtained from the model-guided GA.

Metric	sha	fft	smithwaterman
Max. weighted path	2444	6215	7565
Max. unweighted path	1889	5410	8244
Median of topological count	1444	2351	2765

sha performs reasonably well when the unweighted path metric is used. This indicates that the latency for *sha* has a strong correlation with the length of its critical path. The topological count metric models latency relatively well because it captures information about the structure of the tree and gives an average measure of the variance in ILP available in the program.

To summarize, the models suggested in this initial study see some limited success, but are not sophisticated enough to capture all aspects of a program that collectively affect latency. Building such a model is a non-trivial task and is something that we are considering for future work.

4.3.5 Comparison of best sequences across benchmarks

In this section, we briefly explore the proximity of best sequences of different benchmarks in their impact towards the latency of the RTL generated. In order to measure this, we take the best sequence of each benchmark and run xPilot on every benchmark using that optimization sequence.

Table 4.7 shows the comparison for the five benchmarks using all the best sequences. As can be seen, the best sequence for one benchmark can be the best sequence of another similar benchmark. For example, *matrixmul* performs well on the best sequence for *smithwaterman* as well. This is because they are affected by the same critical optimizations. Similarly, *sha* and *smithwaterman* perform well on each other's best sequences as well.

Table 4.7: Comparison of best sequences. We synthesize each of the four benchmarks using the best optimization sequence from each benchmark. Columns represent benchmarks and rows represent the benchmark whose best sequence is being used. Number reported is latency from xPilot, in cycles.

	matrixmul	sha	smithwaterman	fft	jacobi
bestSeq - matrixmul	49	1526	23	5359	78357
bestSeq - sha	708094	1442	844	2349	62561
bestSeq - smithwaterman	49	1442	23	3359	57206
bestSeq - fft	250676	3982	152152	1896	76781
bestSeq - jacobi	49	1442	23	2359	492

Table 4.8 shows the best sequence for each benchmark. From Tables 4.7 and 4.8, we can notice that the best sequences for *matrixmul* and *smithwaterman* show similar behaviour on all benchmarks, and are hence having the same set of critical optimizations. A closer examination would reveal that the sub-sequence *-mem2reg -loop-preproc -simplifycfg* is present in both sequences, and this sub-sequence happens to be the critical optimization sequence for both *matrixmul* and *smithwaterman*. We re-synthesized the two benchmarks using just the critical sequence to confirm the same. We can also observe in Table 4.7 that the best sequence for *jacobi* performs well on most of the benchmarks. This is because it contains the sub-sequence *-mem2reg -loop-preproc -simplifycfg*, and hence performs better on *matrixmul* and *smithwaterman*. It also contains the optimization *mem2reg* which is the sole critical optimization for *sha* and also greatly affects *fft*. This also explains why the best sequence for *fft* does not perform well on any benchmark - because it does not have the sequence of required critical optimizations.

4.3.6 Comparison with CPU performance

After all the analyses and conclusions, one question still remains. Are the results really unique to High-level synthesis, or does the same sequence qualify to be the best sequence for a general purpose CPU as well? In order to find out, we picked 200 of the randomly

Table 4.8: Best sequences for each individual benchmark.

Benchmark	Best sequence
matrixmul	xunroll -indvars -dse -mem2reg -xunroll -loop-preproc -ptrLegalization -simplifycfg
sha	-instcombine -mem2reg
smithwaterman	gvn -dse -adce -mem2reg -instcombine -adce -loop-preproc -simplifycfg
fft	mem2reg -xunroll -indvars -bitwidthmin -indvars -dse -gvn -ptrLegalization
jacobi	constprop -mem2reg -gvn -simplifycfg -loop-preproc -simplifycfg -bitwidthmin -gcse -simplifycfg -dse -gvn -adce -instcombine -gvn -ptrLegalization

generated optimization sequences for two benchmarks, *sha* and *smithwaterman*. The same sequences were used to compile the same kernel code into object code. The object code was then linked to a driver to create an executable. 200 such executables were created and labelled with their sequence. The executables were simulated on a cycle-accurate out-of-order processor simulator Simics to get accurate cycle counts. The executables were then given a *CPU rank* and an *xPilot* rank based on their execution time and latency respectively, with the lesser quantity obtaining higher rank i.e., the first rank executable has the lowest cycle count.

Figures 4.9 and 4.10 show the rank disparity between xPilot and CPU for the same optimization sequence. In order to better understand this behaviour, let us consider a specific example from benchmark *Sha* in Figure 4.10. Sequence number 1101 has an xPilot rank of 2 while a CPU rank of 175. The optimization sequence 1101 is shown in Figure 4.11.

Further experiments revealed that the HLS-specific *bitwidth* optimization was adding a lot of overhead instructions, thereby increasing the instruction count. While optimizing bitwidth is a very beneficial optimization from a HLS perspective, the side-effects of arbitrary bitwidth-optimized code is the addition of a large number of instructions (like *sethi*, *shl*, *and*

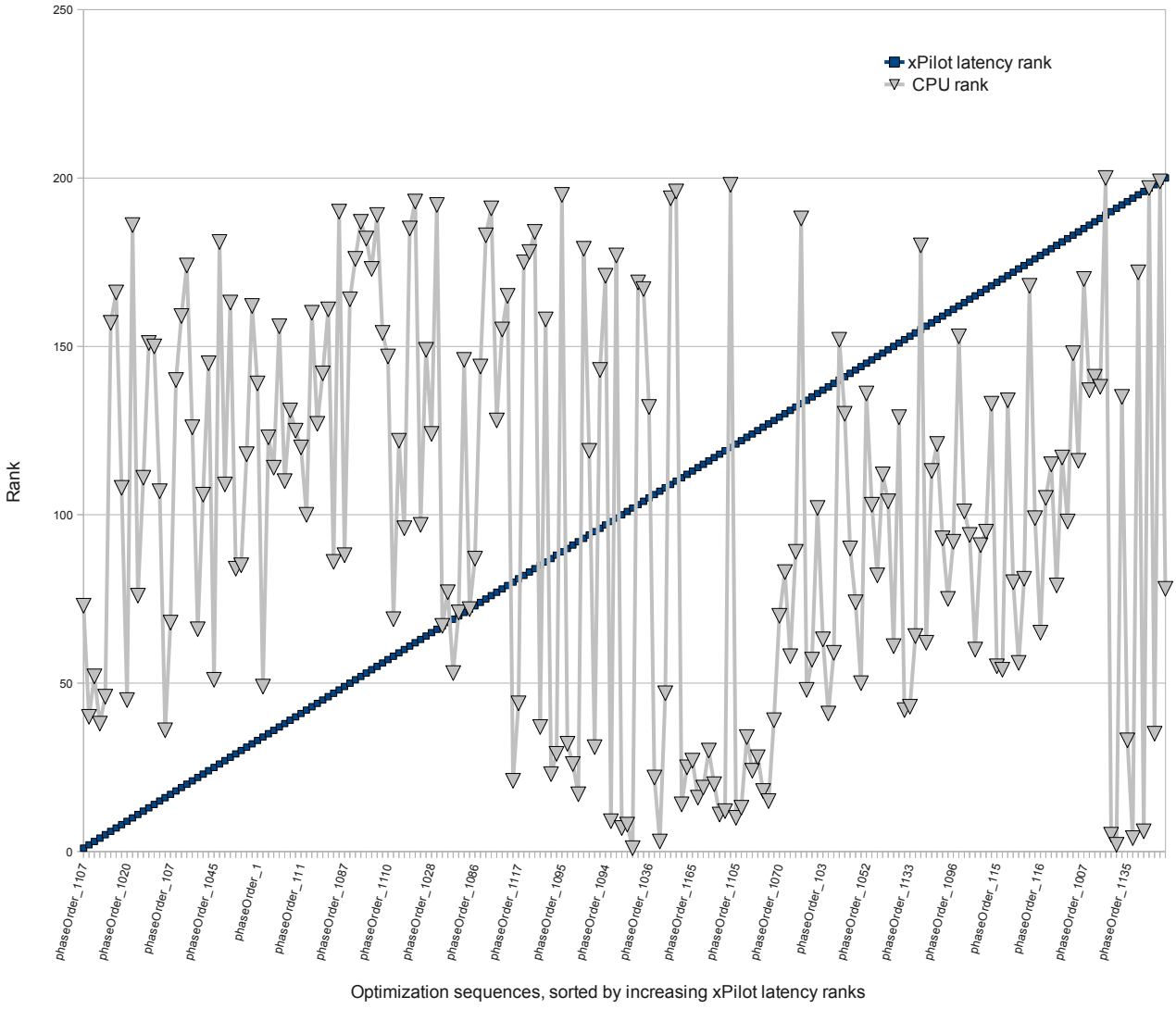

Figure 4.9: Rank comparison for SmithWaterman: CPU vs xPilot. As can be observed, the same sequence ranks differently with respect to other sequences in a CPU setting compared to a HLS setting.

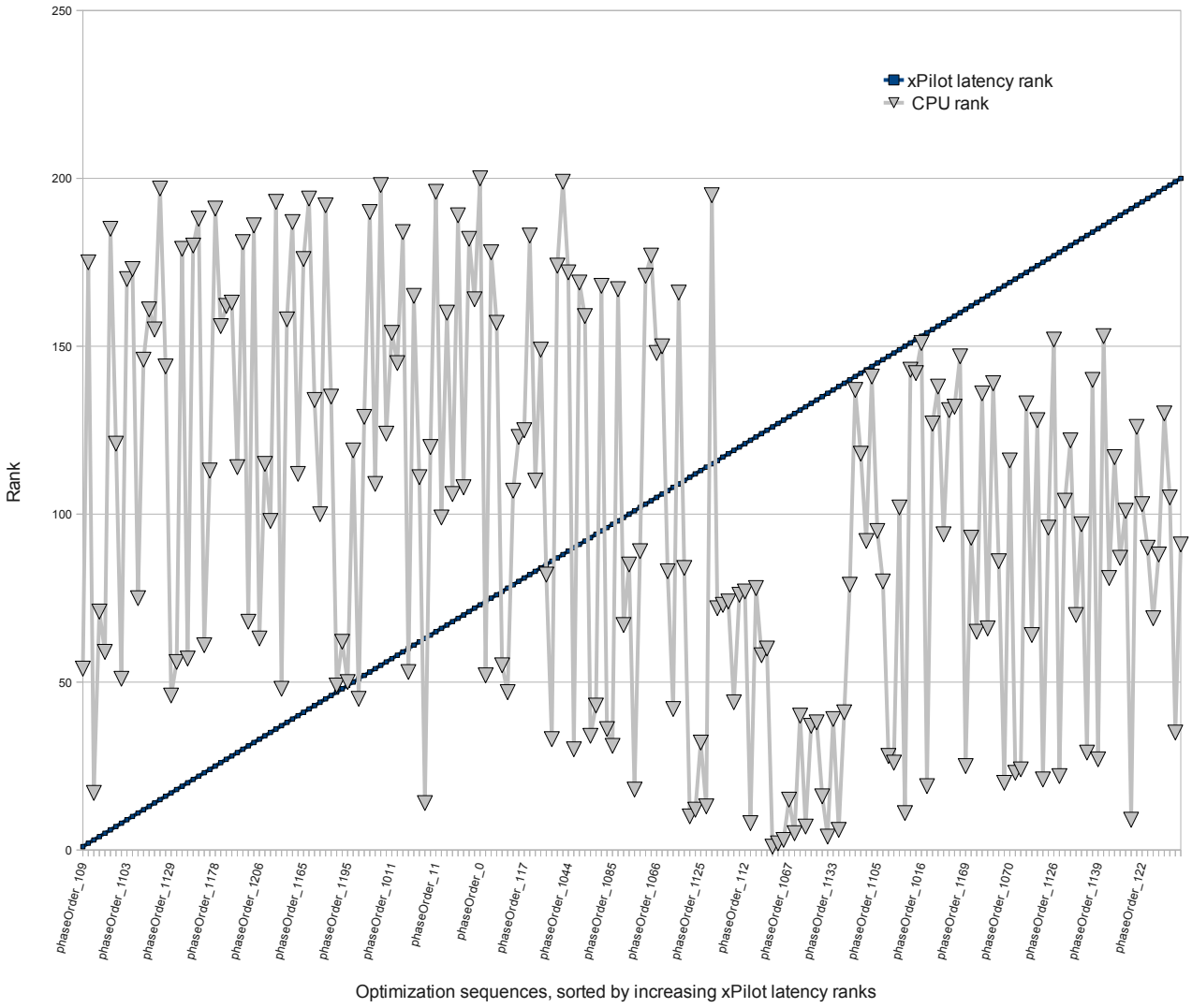

Figure 4.10: Rank comparison for Sha: CPU vs xPilot. As can be observed, the same sequence ranks differently with respect to other sequences in a CPU setting compared to a HLS setting.

```
-loopsimplify -loop-deletion -adce -loop-preproc -mem2reg -bitwidthmin
-loop-preproc -condprop -inst-simplify -gcse -xunroll
-ptrLegalization -condprop
```

Figure 4.11: Optimization sequence 1101 under comparative study between CPU and xPilot. This sequence favours HLS.

```
-gvn -condprop -xunroll -indvars -dse -find-region -array-normalize
-ptrLegalization -indvars -find-region -adce -loop-deletion -constprop
-xunroll -gvn -ptrLegalization -gvn -loopsimplify -xunroll
```

Figure 4.12: Optimization sequence 118 under comparative study between CPU and xPilot. This sequence favours CPUs.

in the Sparc ISA that Simics recognizes) along with dependencies between them. This increases the number of cycles needed to execute the same function. Such side-effects do not exist in HLS because an operator of a specific bit-width can be realized in hardware without additional overhead instructions as needed by a CPU. In order to confirm our suspicions, we generated another binary using the sequence in Figure 4.11, without the *-bitwidthmin* optimization. Table 4.9 shows the difference in the cycle count. We can clearly observe the drop in the cycle count in the case without *-bitwidthmin*.

Table 4.9: CPU cycle count from Simics of two binaries generated using the optimization sequence in Figure 4.11, with and without *-bitwidthmin*.

Binary type	Cycle count from Simics
With *-bitwidthmin*	70790
Without *-bitwidthmin*	55981

Upon further examination, we found that most of the lower-ranked sequences for CPUs had *-bitwidthmin* optimization included in them.

One can argue that the bit-width optimization has always been a HLS-specific optimization and hence it is unfair and unnecessary to study it in a CPU setting. Hence, a more interesting case would be a sequence that suits a CPU better than a HLS setting, which is what we discuss here. Consider Figure 4.12, which shows optimization sequence 118. This sequence has a CPU rank of 9 and an xPilot rank of 191.

Examining this case was much more complicated. It turns out that the pair of optimizations *-gvn* and *-indvars* have an interesting interplay that varies between a CPU setting

and a HLS setting. To study this in greater detail, we generated binaries for several cases and compared the behaviour of the CPU and HLS setting. Table 4.10 summarizes our experiments.

Table 4.10: Comparison of CPU and HLS settings with optimization sequences involving *-gvn* and *-indvars*.

Optimization sequence	xPilot latency	CPU cycle count
(none)	1844	54710
-indvars	1844	54709
-indvars -gvn	1444	54959
-gvn	1444	54958
-gvn -indvars	4024	53644

We can see that the pair *-gvn* and *-indvars* affect the CPU and HLS setting in opposite ways. First, we note that as the benchmark loops are in normal form by default, *-indvars* alone does not have any effect, and-*gvn* creates opportunity for *-indvars*. The *-gvn* optimization tries to remove redundant instructions and tries to re-use previously computed values. Hence, intuitively *-gvn* decreases code size, and our simulation runs confirm that. The fact the *-gvn* increases CPU cycle count in spite of reducing the number of instructions might seem puzzling at first, but can be explained as follows. While *-gvn* removes redundant code, it can have a potential side effect of introducing data dependency due to re-use. Also, if the data to be re-used is in memory, *-gvn* can slightly increase the number of loads in the program, as is the case in our example. The combined effect leads to an increased number of pipeline stalls which explains the increased cycle count. The *-indvars* pass adds simple additional instructions like add, sub etc to canonicalize loop induction variables, and hence increases code size. Also, the implementation of *-indvars* tends to promote certain memory values to registers, thereby reducing the number of loads. Running an *-indvars* pass after *-gvn* pass increases the number of instructions, reduces the number of loads and pipeline stalls, and thus effectively un-does the damage caused by *-gvn*. Hence we see that in the CPU setting, *-indvars* has a positive effect after *-gvn*.

In the HLS setting, however, there is no instruction execution pipeline. The design can be seen as a data and control-flow driven application where a finite set of instructions can be scheduled to run in every cycle. Hence, fewer instructions need fewer cycles to run. This explains the positive effect of *-gvn* on the xPilot latency. Running an *-indvars* pass after *-gvn* introduces many additional instructions which are dependent on existing instructions. This leads to an increase in the number of states in the FSM and an increase in latency.

We re-ran xPilot using the sequence in Figure 4.12 without the *-indvars* option and observed that the latency dropped from 4024 to 1444.

The above experiments show that a good sequence for a CPU need not necessarily be a good sequence for a hardware designer aiming to reduce latency. This finding also provides empirical proof that the best sequences cannot be blindly borrowed using results from a processor.

CHAPTER 5

Related Work

Numerous approaches have been proposed by researchers to explore the compiler optimization solution space, and they can be broadly classified into the following groups:

- Design space exploration: So et al [23] use synthesis estimation techniques to evaluate design alternatives for a loop nest. They introduce a *balance* metric that represents the computation and memory requirements of a loop nest, based on which the loop is classified to be either compute-bound or memory bound. The designs are optimized for performance. The authors in [21] develop a mathematical model that captures various loop characteristics like parallelism, dependencies and resource requirements. This model is used to find an optimal unroll factor. Here again, the optimal unroll factor is the one that achieves best performance under a given resource constraint. The model along with an FPGA frequency estimation technique is used to analyze whether loop unrolling alone is a better optimization choice or unrolling combined with software pipelining. Other works like [11] focus on studying the effects between loop unrolling and shifting on loops that are mapped onto re-configurable fabric. This approach uses profiling information obtained by executing the code on a general-purpose processor to determine the most appropriate transformation. Studies like [28] use iterative compilation to study the tradeoff between software pipelining and loop unrolling. [3] presents a survey of approaches taken to model loop unrolling along with issues that are still open to be addressed.

- Searching techniques: This body of work treats optimizations as black-boxes and uses searching techniques like genetic algorithms, hill climbing etc., to minimize the cost

function. Cooper et al [10] used genetic algorithms to find an optimization sequence that gives the smallest code. They chose a sequence of length 12 from a set of ten optimizations using the static length of the code emitted as the cost function to be minimized. Several techniques have been proposed to speed up the time to convergence of the searching algorithms. Cooper et al suggest a *virtual execution* [9] technique where the program is profiled just once, and structural changes made to the program by transformations are used to estimate the change in latency. Kulkarni et al [17] suggest a different route for faster convergence, by performing a series of checks to reduce redundant execution during searches, as well as providing more intelligence to the search algorithm itself - by the initial population choice, past history of interacting optimizations and so on. Kisuki et al [16] evaluate a variety of searching techniques to select the best combination of tile size and unroll factor.

- Statistical and machine learning techniques: Stephenson et al [25] use an unsupervised learning technique in the form of genetic programming to tune optimization heuristics (like savings on spilling a variable) for optimizations like register allocation, hyperblock formation etc on a per-program basis. Supervised classification techniques like *support vector machines* and *nearest neighbor* have been used in [24] to predict optimal or near-to-optimal loop unroll factors. Agakov et al [2] use supervised classification to identify the probabilities of optimizations to be "good" for a given input program. These probabilities are then used to make choices in search algorithms to achieve faster convergence.

- Exploration techniques: Triantafyllis et al propose OSE [27], where a set of optimizations are selected at compiler construction time using a "compiler construction pruning" algorithm. A tree of optimizations is built out of the selected set, which is traversed to select the best optimization order for a given program. Pan et al [22] propose algorithms like Batch Elimination, Iterative Elimination and Combined Elimination which attempt to find a set of optimizations that minimize execution time. However, this work does not consider the order of optimizations. A more recent work

by Chabbi et al [4] proposed an interesting idea of exploring optimization sequences one pair at a time, and is close to some of the analyses that we have performed.

- Model-based approaches: Whitfield et al [29] [30] proposed a specification language to describe the nature of each optimization as aset of entry conditions, transformation actions and exit conditions. By having such a specification for each optimization, automated detection of enabling and disabling conditions between optimizations can be performed. [33] extends [30] to include a larger class of optimizations. Zhao et al use a combination of code, optimization and resource models to predict the impact of high-level optimizations such as loop unrolling, loop tiling etc [31] as well as scalar optimizations such as partial redundancy elimination and register allocation [32]. In our work so far, we have used a simple model with a few simple features and we have not modeled resource constraints. We are looking into the same as part of future work. Tate et al [26] propose a method of "equality saturation", where each optimization does not transform a program but adds *equality information* to the IR. Such an IR maintains multiple versions of the source program, and an optimal version can be chosen by just picking among equivalent nodes based on a certain cost function. A hybrid method of empirical approach and analytic models [12] has been used to tune the performance of libraries, such as ATLAS, on a particular target.

As evident from above, there is a vast literature of work in this broad area. However, each contribution is targeted towards a specific subset of the problem, and hence the area of design space exploration using compiler optimization phase ordering is still far from being solved. Our work here uses past analyses to quickly eliminate bad choices so that the right optimization metrics are arrived at quickly.

CHAPTER 6

Conclusions

Given the rise in popularity in high-level synthesis as a popular design choice in the system design community, we believe that having a sound compilation technology in high-level synthesis is very essential. In this paper, we have presented a first study on the impact of compiler optimization phase ordering on design space exploration and the quality of the generated RTLs. Several differences exist between a CPU-based setting and a High-level synthesis design from a compilation perspective. For instance, there is no branch prediction or speculative execution, so the execution time savings achieved by such hardware enhancements are not applicable in a HLS domain. Absence of caches means that each load costs the same, and can only be amortized by greater ILP. HLS tools can exploit greater ILP constrained only by the resources on the target platform, while an out-of-order processor can exploit ILP only among instructions in its scheduling window. We provided a strong motivation to our study using using simple pedagogical examples as well as data collected from well-known and commercial tools like AutoESL and Simics. We studied the impact of high-level optimizations and IR-level optimizations using AutoESL and xPilot respectively. AutoESL is a commercial high-level synthesis tool which provides an easy interface to specify high-level optimizations. However, AutoESL does not give the user the capability to specify lower-level optimizations. Hence, we evaluate IR-level optimizations using xPilot.

We have presented studies on three important high-level optimizations - loop unrolling, software pipelining and array partitioning and interactions between them. We demonstrate and provide insights into the effects of these optimizations using simple examples and devise a strategy to cover the search space. We show and explain that the best partitions are always power-of-2 factors so that index calculation is cheaper. We also show that the best unroll

factors depends largely on the loop trip-count and form a restricted set of unroll factors using powers-of-2 and factors of the trip-count. We use simple heuristics to quickly eliminate bad choices early and use existing estimation techniques in AutoESL to arrive at an unroll factor that provides a good performance to area. We present studies on the combined effect of the three optimizations and demonstrated its effectiveness on several benchmarks. We use the *efficiency ratio* or *ER*, which is the ratio of the latency-area product to evaluate the optimized benchmarks against the baselines. We achieve a maximum ER of 11.29, which shows that significant enhancements can be achieved with the right combination of the three optimizations. At the same time, we also demonstrate cases where the ER is less than 1, and discuss the reasons and pitfalls.

We also presented a detailed study of the effects of scalar optimization phase ordering on high-level synthesis. We have shown the discreteness and step-constant nature of the optimization solution space. We have then discussed and evaluated a variety of techniques like random search and genetic algorithm. We have presented a general n-lookahead technique and show that a 1-lookahead can achieve the same level of performance as random search in majority of cases. We have extracted and evaluated a few simple latency estimation metrics based on static program features, but with limited success. We have reported a mean reduction in latency of 50.42% against xPilot's default setting. We have also studied the generality in the best sequences of optimizations and found that the best sequence for one benchmark can indeed be the best sequence for other benchmarks that are similar. We compare our study to a CPU-based setting and provide several insights into the subtle variations that causes pairs of optimizations to have different effects.

We believe that our work opens up many interesting future directions. An interesting direction for future work would be to consider more high-level optimizations like loop tiling, strip mining etc. Such a design space is extremely vast, and accurate models of representation is required for both the code and the optimizations themselves. We believe that it would be beneficial to have a more sophisticated estimator that would provide estimations with greater accuracy for any arbitrarily complicated program. With the vast amount of data that we have collected, one of our future directions is to explore the idea of building a predictive

model using static program features. Yet another potential direction could be to explore the idea of capturing the behavior of compiler optimizations using latency-sensitive or area-sensitive program features. With our promising initial results, we believe that research in this direction would benefit the high-level synthesis community.

References

[1] Open Source Accelerator Store http://cadlab.cs.ucla.edu/accelerator_store.html.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[3] J. Cardoso and P. Diniz. Modeling loop unrolling: Approaches and open issues. In A. Pimentel and S. Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation*, volume 3133 of *Lecture Notes in Computer Science*, pages 224–233. Springer Berlin Heidelberg, 2004.

[4] M. M. Chabbi, J. M. Mellor-Crummey, and K. D. Cooper. Efficiently exploring compiler optimization sequences with pairwise pruning. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 34–45, New York, NY, USA, 2011. ACM.

[5] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *Proc. IEEE Int. SOC Conf.*, pages 199–202, 2006.

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[7] J. Cong, B. Liu, and J. Xu. Coordinated resource optimization in behavioral synthesis. In *Proc. Design, Automation and Test in Europe*, pages 1267–1272, 2010.

[8] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proc. Design Automation Conf.*, pages 433–438, 2006.

[9] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '05, pages 69–77, New York, NY, USA, 2005. ACM.

[10] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, LCTES '99, pages 1–9, New York, NY, USA, 1999. ACM.

[11] O. S. Dragomir, T. Stefanov, and K. Bertels. Optimal loop unrolling and shifting for reconfigurable architectures. *ACM Trans. Reconfigurable Technol. Syst.*, 2(4):25:1–25:24, Sept. 2009.

[12] A. Epshteyn, M. J. Garzaran, G. DeJong, D. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali. Analytic models and empirical search: a hybrid approach to code optimization. In *Proceedings of the 18th international conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 259–273, Berlin, Heidelberg, 2006. Springer-Verlag.

[13] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin. *High-level synthesis: introduction to chip and system design*. Kluwer Academic Publishers, 1992.

[14] R. Gupta and F. Brewer. *High-Level Synthesis: A Retrospective*, pages 13–28. Springer, 2008.

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[16] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, PACT '00, pages 237–, Washington, DC, USA, 2000. IEEE Computer Society.

[17] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 171–182, New York, NY, USA, 2004. ACM.

[18] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Exhaustive optimization phase order space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 306–318, Washington, DC, USA, 2006. IEEE Computer Society.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. on Code Generation and Optimization*, page 75, 2004.

[20] B. Lee. A new algorithm to compute the discrete cosine transform. *IEEE Trans. Acoustics, Speech and Signal Processing*, (6):1243–1245, Dec. 1984.

[21] J. Liao, W.-F. Wong, and T. Mitra. A model for hardware realization of kernel loops. In P. Y. K. Cheung and G. Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 334–344. Springer Berlin / Heidelberg, 2003.

[22] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 319–332, Washington, DC, USA, 2006. IEEE Computer Society.

[23] B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast hardware design space exploration in fpga-based systems. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 165–176, New York, NY, USA, 2002. ACM.

[24] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[25] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 77–90, New York, NY, USA, 2003. ACM.

[26] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM.

[27] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[28] P. van der Mark, E. Rohou, and C. Eisenbeis. Using iterative compilation for managing software pipeline-unrolling trade-offs, 1999.

[29] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, PPOPP '90, pages 137–146, New York, NY, USA, 1990. ACM.

[30] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19:1053–1084, November 1997.

[31] M. Zhao, B. Childers, and M. L. Soffa. Predicting the impact of optimizations for embedded systems. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 1–11, New York, NY, USA, 2003. ACM.

[32] M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: An approach for profit-driven optimization. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 317–327, Washington, DC, USA, 2005. IEEE Computer Society.

[33] M. Zhao, B. R. Childers, and M. L. Soffa. A framework for exploring optimization properties. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 32–47, Berlin, Heidelberg, 2009. Springer-Verlag.