# A Model for Hardware Realization of Kernel Loops

Jirong Liao, Weng-Fai Wong, and Tulika Mitra

Department of Computer Science, School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543
{liaojiro, wongwf, tulika}@comp.nus.edu.sg

**Abstract.** Hardware realization of kernel loops holds the promise of accelerating the overall application performance and is therefore an important part of the synthesis process. In this paper, we consider two important loop optimization techniques, namely loop unrolling and software pipelining that can impact the performance and cost of the synthesized hardware. We propose a novel model that accounts for various characteristics of a loop, including dependencies, parallelism and resource requirement, as well as certain high level constraints of the implementation platform. Using this model, we are able to deduce the optimal unroll factor and technique for achieving the best performance given a fixed resource budget. The model was verified using a compiler-based FPGA synthesis framework on a number of kernel loops. We believe that our model is general and applicable to other synthesis frameworks, and will help reduce the time for design space exploration.

## 1 Introduction

A standard practice in synthesis of application specific hardware is to focus attention at kernel loops. In many applications, they account for the bulk of the execution time and are thus natural candidates for hardware acceleration. A key difficulty in synthesizing hardware for kernel loops is that there are many loop optimizations available and the complex interactions among these optimizations make it difficult to predict the cost-benefit of applying each. In particular, one cannot tell how much more or less resources a particular optimization will take or what its impact will be on performance. This means that one has to either settle for sub-optimal results or go through a costly process of trial-and-error in order to arrive at the correct combination of loop optimizations that fits the need of the user. Having a model of how a particular loop optimization will impact resource and performance is therefore necessary.

Two important loop optimizations applicable to kernel loops are *loop unrolling* and *software pipelining*. Loop unrolling is a technique to expand the loop such that a new iteration consists of 2 or more of the original iterations. This is performed by a compiler to expose more instruction level parallelism and reduce the overhead of updating index variables. The number of times the loop is expanded is called the *unroll factor*. If the loop iteration count is not a multiple of unroll factor, then the remainder of the loop iterations needs to be executed at the end as it is.

Software pipelining [1] tries to achieve higher level of instruction level parallelism by moving operations across iteration boundaries. This optimization achieves overlap among the iterations by pipelining the execution of the iterations. The loop body is scheduled such that (a) all iterations have identical schedule and (b) each iteration is scheduled to start some fixed number of cycles later than the previous iteration. The delay between the start cycles of two successive iterations is called the *Initiation Interval (II)*. The modulo scheduling algorithm attempts to achieve the smallest value of II such that no intra- or inter-iteration dependencies and resource constraints are violated.

As multiple iterations are executed in parallel, both loop unrolling and software pipelining increase register pressure and resource requirement but in different ways. Furthermore, it is possible to use them in combination, i.e. it is possible to software pipeline unrolled loops. The complex interaction between the two optimizations makes it difficult to decide how they should be deployed in optimizing a loop given a particular resource constraint. Often the only way to tell is to exhaustively try various combinations of these two optimizations to obtain the optimal one.

In this paper, we propose a model for the performance and resource requirement for the hardware realization of unrolled and software pipelined loops. The novelty of our model lies in the use of the compiler to extract certain key parameters of the loop in question that characterize the code including the data dependences present for a given hardware. For example, the platform we use allows at most four parallel reads to memory and only if they do not hit the same memory bank. Such characteristics are hard to model. So instead we rely on the instruction scheduler of a compiler to capture these. From these parameters reported by the compiler, the model will inform the user if given a certain resource constraint, unrolling alone, or software pipelining used in combination with loop unrolling would deliver the better performance. It will also output the optimal unrolling factor that should be used. The contribution of this model is that without exhaustively trying a large number of possibilities, it can very quickly recommend a solution that we believe is optimal or very near it.

## 2 Related Work

Hardware realization of kernel loops has been actively studied by many research groups. However, the focus has been mainly on automatic synthesis of kernel loops from high level language constructs. The exploitation of compiler optimizations such as loop unrolling and modulo scheduling has largely remained unexplored. Even a few commercial synthesis tools that apply these compiler optimizations depend on user feedback to choose unroll factor or decide between unrolling and modulo scheduling. Our work bridges this gap in automatic hardware realization of kernel loops.

There are two main approaches towards hardware synthesis from high level constructs. One approach is to design new languages for hardware design which are at much higher level than traditional hardware description languages such as Verilog and VHDL. The claim is that the productivity gap will be reduced as software programmers can easily learn these new languages. An example is Handel-C [2] programming language which has C-like syntax with support for explicit hardware parallelism, communication, and hardware structures such as memory, bus etc.

The other approach attempts to map a subset of commonly used software programming languages such as C to hardware automatically. These efforts include SA-C [3], PipeRench C Compiler [4], Garp C compiler [5], work by Weinhardt et. al. [6] [7], Babb et. al. [8] and Snider et. al. [9]. The PACT project [10] at Northwestern University performs C to hardware synthesis by taking power/performance trade off into account. The PICO project [11, 12] performs static timing analysis to identify chain of operators to minimize number of cycles while maintaining cycle time constraints.

The only existing tool that allows application of high level compiler optimizations in hardware synthesis is Monet [13]. However, it requires user feedback in deciding unroll factor for example. Among research projects, Derien et. al [14] have developed an analytical model to choose a tiling strategy that will minimize loop execution time. The closest to our work is So et. al. [15]. They perform fast and automatic design space exploration to choose the right loop unrolling factor that satisfies the area constraints and maximizes performance. However, they do not use other compiler optimizations such as software pipeline which can potentially improve the performance significantly.

## 3 Our Model

In this section, we will present our proposed model. The novelty of the model lies in the use of key parameters supplied by the compiler in characterizing aspects of the kernel loop as well as the machine that are hard to model correctly.

### 3.1 Model for Performance

For the discussion below, we will assume a loop $L$ that is executed $N$ times. Let $S_1$ be the *schedule length* of the loop. In our model, $S_1$ is a quantity reported by the compiler as it performs instruction scheduling. As we are realizing the loop in hardware, we assumed infinite registers by skipping the traditional register allocation phase. In the quantity $S_1$, various complex issues such as the machine's configuration, instruction type distribution, data dependencies etc. are encapsulated. The user, for example, can choose to use the machine configuration to constraint the amount of parallelism or number and types of functional units to be realized in hardware. We will also generalize $S_1$ to $S_u$ which is the schedule length of the kernel when it is unrolled $u$ times. The following formula gives the total number of cycles the unrolled kernel will take to execute $N$ iterations.

$$C_{\text{unrolled}}(u) \approx \left\lfloor \frac{N}{u} \right\rfloor \times S_u + \left( N - \left\lfloor \frac{N}{u} \right\rfloor \times u \right) \times S_1 \tag{1}$$

After unrolling, the loop size is $\lfloor N/u \rfloor$ and the schedule length is $S_u$. Therefore the first term in Eq. 1 accounts for the total number of cycles executed by the unrolled loop. However, if $N$ is not divisible by $u$, a compensation loop of size $N - \lfloor N/u \rfloor \times u$ and a schedule length of $S_1$ will be generated. In practice, we would not want to have to get all $S_u$'s from the compiler as that requires multiple runs. Rather, we estimate $S_u$ given $S_1$. In particular, we assumed that

$$S_u = S_{u-1} + c_S \tag{2}$$

where $c_S$ is a constant. From the experience gained from our experimentation, we chose

$$c_S = \frac{(S_3 - S_1)}{2}$$

This is because we found that there may be a case where it so happens that empty resource slots available at the end of the instruction schedule can be filled up by a new instance of the loop.

To model software pipelining, we assumed the technique of *iterative modulo scheduling* given by Rau [1] that uses *predicated execution* and *rotating registers* [16]. It is characterized by two important parameters also obtained from the compiler, the initiation interval, $II$, and the epilog counter $e$. The initiation interval is the gap (in machine cycles) between two successive software pipelined iterations. In effect, after a successful modulo scheduling, each iteration of the software pipelined kernel loop takes exactly $II$ cycles. The epilog count is the number of iterations in the epilog of the software pipelined loop. Again, in $II$ and $e$, the complexity of machine configuration, resource requirements, and data dependencies are hidden away. Since we would like to combine software pipelining with unrolling, we will introduce $II_u$ and $e_u$ which are the $II$ and $e$ for a software pipelined loop that has been unrolled $u$ times. We have the following formula for the total number of cycles a software pipelined loop that has been previous unrolled $u$ times will take:

$$C_{\text{swp}}(u) \approx \left( \left\lfloor \frac{N}{u} \right\rfloor + e_u \right) \times (II_u + 1) + 3 + \left( N - \left\lfloor \frac{N}{u} \right\rfloor \times u \right) \times S_1 \tag{3}$$

A constant of 1 is added to $II_u$ because at the end of each iteration, it is necessary to perform a shift of the content of the rotating registers so as to prepare for the next iteration. These shifts can be done in parallel in hardware and thus cost one cycle. The constant of 3 is needed because in our scheme, we needed one clock cycle at the beginning of the loop to set up the rotating registers, another clock cycle to initiate the loop and epilog counters, and one more at the end of the loop to copy out the content of the rotating registers.

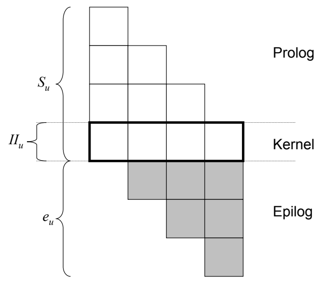

**Fig. 1.** Relationship between $S_u$, $II_u$ and $e_u$.

$S_u$ is obtained from Eq. 2. As is the case for $S_u$, we do not redo modulo scheduling over all possible $u$'s for $II_u$ and $e_u$. Given a machine configuration, $M$, and a loop, $L$, the following holds:

$$II_u = II_{u-1} + c_{II} \tag{4}$$

$$e_u = \left\lceil \frac{S_u}{II_u} \right\rceil - 1 \tag{5}$$

where $c_{II}$ is dependent on $M$ and $L$. However, we also found that the simple recurrent relation for $II_u$ do not necessarily end with the unroll size of 1. In particular, for software pipelining, if there is sufficient resources, then $II_i = II_{i-1}$ and the recurrent relations are not established until resource over-subscription comes into play. In our experiments, we used a machine that has only four memory port but otherwise has unlimited resources. The former condition is to reflect the limitation of the FPGA board that we are using. We used the following strategy: we perform software pipelining with $II_1, II_2, \ldots$ until $II_i \neq II_{i-1}$.

$e_u$ can be derived from $S_u$ and $II_u$ through Eq. 5. This relationship is apparent once we see the idealized diagram for software pipelining shown in Fig. 1. In this example, $S_u = 4$, and $II_u = 1$, giving $e_u = 3$. Since $S_u > II_u$, $e_u \geq 1$.

**Estimating FPGA Frequencies.** The total running time of an implementation of a loop in a FPGA is given by the product of the number of cycles it takes to execute the code and the frequency of the FPGA which permits the safe operation of the realized design. It turns out that it is difficult to use static compiler information to obtain an accurate model of the final realizable frequency. In order to overcome this problem, we use the following strategy. We run place and route for three instances of the loop, namely the loop unrolled two, three, and four times. These three runs are also used in our resource estimation process described in the next section. Let the actual frequencies obtained from the three runs be $f_l(2)$, $f_l(3)$ and $f_l(4)$, respectively where $l$ is either 'unrolled' or 'swp'. We set the predicted frequency as follows:

$$F_l(u) = \begin{cases} \max(f_l(2), f_l(3), f_l(4)) & \text{if } u = 1 \\ f_l(u) & \text{if } u = 2, 3, \text{ or } 4 \\ \min(f_l(2), f_l(3), f_l(4)) & \text{if } u > 4 \end{cases} \tag{6}$$

Using these equations, we can finally approximate the time taken to execute the realized design to be

$$T_{\text{unrolled}}(u_1) = C_{\text{unrolled}}(u_1) \times F_{\text{unrolled}}(u_1) \quad \text{and} \quad T_{\text{swp}}(u_2) = C_{\text{swp}}(u_2) \times F_{\text{swp}}(u_2)$$

## 3.2 Model for Resource Usage

While we can easily count the various operators emitted by the compiler, optimizations further down the synthesis chain, in particular, the place and route pass, introduce non-trivial relationships between the high level hardware description our compiler output and the final resource usage. From experimental results, we found this to be especially true for the case of software pipelined loops. From the same three place and route runs used to obtain the frequencies, we also obtained the resource consumption information by means of linear regression. In particular, for a machine $M$ and loop $L$, we model resource usage as:

$$R_{\text{unrolled}}(u) = m_{\text{unrolled}} \times u + c_{\text{unrolled}} \tag{7}$$

$$R_{\text{swp}}(u) = m_{\text{swp}} \times u + c_{\text{swp}} \tag{8}$$

where $m_{\text{unrolled}}$, $c_{\text{unrolled}}$, $m_{\text{swp}}$, and $c_{\text{swp}}$ are constants obtained from the linear regression.

### 3.3 Putting it together

The model is used as follows. The user will decide on a certain amount of resource, $R_{\text{user}}$, that he would like to use for realizing the loop in hardware. Using Equations 7 and 8, we obtained two maximal unroll factors $u_1$ and $u_2$ such that

$$R_{\text{unrolled}}(u_1) \leq R_{\text{user}} \qquad \text{and} \qquad R_{\text{swp}}(u_2) \leq R_{\text{user}}$$

Next we examine all unroll factors less than $u_1$ and $u_2$ to look for a $u'_1 \leq u_1$, and a $u'_2 \leq u_2$ such that $T_{\text{unrolled}}(u'_1)$ and $T_{\text{swp}}(u'_2)$ are the respective minimum. If $T_{\text{unrolled}}(u'_1) > T_{\text{swp}}(u'_2)$ then we will get better performance by using software pipelining with the loop unrolled $u'_2$ times and vice versa.

## 4 Compilation Framework

We used the Trimaran [17] compiler infrastructure to experiment with the model. The compiler targets for a parameterized Explicitly Parallel Instruction Computing (EPIC) architecture called HPL-PD [16]. We modified the compilation framework as follows:

- An EPIC machine with infinite resources except for four memory ports was defined. The four memory port was a constraint of the FPGA board which we used in our experiments. It has four banks of memory that can be simultaneously accessed with only one access to a bank at any time. Consequently, we also had to modify the instruction and modulo schedulers of Trimaran. We assumed that an entire array is stored in a single bank. Thus any two access to the same array has to be performed in different machine cycles.
- Trimaran uses some heuristics to guide unrolling. Furthermore, it does not always emit compensation loops during unrolling as these can be folded into the unrolled loop using predicated execution. For our purpose, we forced unrolling to be performed as per our requirements.
- Finally, we added a phase to generate Handel-C [18] code for Trimaran's Elcor intermediate representation. Handel-C is a C-like behavioral hardware description language. The Handel-C compiler compiles our output into a EDIF [19] file for the FPGA vendor's synthesis tools to process.

In the resultant design flow, we are able to utilize the advanced features used by Trimaran including predicated execution and rotating registers and translate them into Handel-C. From Handel-C's EDIF output, we synthesis the bitmap for a Xilinx XCV1000 FPGA and execute it on a Celoxica RC1000 board.

## 5 Results

We used six kernel loops to verify our model:

- **Edge detection.** A $32 \times 32$ mask is computed over $128 \times 128$ image to detect edges.
- **Matrix multiplication.** Integer multiplication of $160 \times 320$ and $320 \times 40$ matrix.
- **Finite impulse response filter.** A 128-tap FIR filter on 256 integer data values.
- **Livermore Loop 1.** Hydro fragment loop of size 1001.
- **Jacobi.** 4-point stencil averaging computation over an array with loop size of 100.
- **Histogram.** Mapping from the old to the new grey levels with loop size of 1024.

The accuracy of our performance model is given in Table 1. The first set of columns present the result for loop unrolling and the second set of columns present the result for unrolling and software pipelining. "Est." is the predicted execution time, i.e. $T_{\text{unrolled}}(u)$ and $T_{\text{swp}}(u)$. "Act." is the actual execution time taken to execute the loop. This is obtained from multiplying the actual frequency obtained after place and route with the actual number of cycles executed. "Diff_T" represents the percentage difference between "Est." and "Act." while "Diff_C" represents the percentage difference in estimating $C_{\text{unrolled}}(u)$ and $C_{\text{swp}}(u)$. The average value for "Diff_C" for loop unrolling and loop unrolling with modulo scheduling are 2.84% and 2.19%, respectively. In addition, the values for $S_u^p, II_u^p$ and $e_u^p$ in Table 1 were computed using Equations 2, 4 and 5 while $S_u^a$, $II_u^a$ and $e_u^a$ were obtained from the actual compilation. The average relative error for "Diff_T" are 3.6% and 8.4% respectively for loop unrolling alone and software pipelining with unrolling. Given that the average relative difference between the actual execution time of the two strategies is 36%, we conclude that our performance estimation model is within the necessary margin and is accurate.

Fig. 2 shows the accuracy of our resource model. Due to space limitation, we will show the results for two benchmarks: Edge and LM1. The results for other benchmarks are similar. "Unroll" and "SWP" show the actual resource usage due to unroll and unroll with software pipeline respectively. These points are obtained from the reports of the FPGA synthesis tool. The "Linear of Unroll" and "Linear of SWP" show the estimated resource usage using linear regression of $u = 2, 3$ and 4. As can be seen from the figures, the estimated resource usage closely follows the actual resource usage.

It seem that in most cases, unrolling alone yields better performance under the same resource constraints. However, if we set $R_{\text{user}} = 100,000$, then for the Lm1 benchmark, the unroll factor to be used for unrolling and software pipelining are 7 and 5, respectively. Using these unroll factors, our model predicts that we should use software pipelining instead of unrolling. The actual execution time given in Table 1 confirms that our prediction is correct.

Table 2 shows the various constants of Equations 7 and 8 obtained in our model. The results show that our model is fairly accurate and can significantly cut down the design space exploration time.

| **Benchmark** | | **Unrolling Only** | | | | | | **Unrolling + SWP** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $(u)$ | $S_u^p$ | $S_u^a$ | **Est. msec** | **Act. msec** | **Diff_T (%)** | **Diff_C (%)** | $II_u^p$ | $II_u^a$ | $e_u^p$ | $e_u^a$ | **Est. msec** | **Act. msec** | **Diff_T (%)** | **Diff_C (%)** |
| | 1 | 4 | 4 | 0.421 | 0.391 | 7.46 | -2.38 | 1 | 1 | 3 | 3 | 0.221 | 0.223 | -1.09 | -4.11 |
| | 2 | 5 | 4 | 0.282 | 0.296 | -4.90 | -4.90 | 2 | 2 | 2 | 2 | 0.177 | 0.189 | -6.75 | -6.75 |
| | 3 | 6 | 6 | 0.227 | 0.244 | -7.01 | -7.01 | 3 | 3 | 1 | 1 | 0.172 | 0.188 | -8.52 | -8.52 |
| Edge | 4 | 7 | 7 | 0.184 | 0.201 | -8.38 | -8.38 | 4 | 4 | 1 | 1 | 0.145 | 0.161 | -9.65 | -9.65 |
| | 5 | 8 | 8 | 0.197 | 0.207 | -5.00 | -9.86 | 5 | 5 | 1 | 1 | 0.166 | 0.178 | -7.05 | -8.82 |
| | 6 | 9 | 9 | 0.187 | 0.201 | -7.25 | -10.36 | 6 | 6 | 1 | 1 | 0.166 | 0.198 | -16.46 | -7.22 |
| | 7 | 10 | 10 | 0.197 | 0.199 | -0.77 | -9.86 | 7 | 7 | 1 | 1 | 0.184 | 0.218 | -15.37 | -5.03 |
| | 8 | 11 | 11 | 0.155 | 0.158 | -1.68 | -12.22 | 8 | 8 | 1 | 1 | 0.15 | 0.226 | -33.49 | -4.24 |
| | 1 | 4 | 4 | 731.5 | 743.4 | -1.60 | -0.16 | 1 | 1 | 3 | 3 | 411.0 | 382.2 | 7.53 | -0.47 |
| | 2 | 5 | 5 | 483.4 | 485.2 | -0.38 | -0.34 | 2 | 2 | 2 | 2 | 318.8 | 322.1 | -1.03 | -1.03 |
| | 3 | 6 | 6 | 368.0 | 370.3 | -0.62 | -0.63 | 3 | 3 | 1 | 1 | 291.2 | 295.2 | -1.36 | -1.36 |
| MM | 4 | 7 | 7 | 361.2 | 363.8 | -0.72 | -0.72 | 4 | 4 | 1 | 1 | 258.4 | 262.2 | -1.47 | -1.47 |
| | 5 | 8 | 8 | 330.3 | 308.6 | 7.00 | -0.78 | 5 | 5 | 1 | 1 | 260.7 | 281.8 | -7.50 | -1.52 |
| | 6 | 9 | 9 | 312.8 | 289.8 | 7.97 | -1.23 | 6 | 6 | 1 | 1 | 258.0 | 298.5 | -13.54 | -1.54 |
| | 7 | 10 | 10 | 303.2 | 275.0 | 10.24 | -1.27 | 7 | 7 | 1 | 1 | 253.4 | 290.3 | -12.70 | -1.57 |
| | 8 | 11 | 11 | 283.8 | 283.5 | 0.12 | -1.36 | 8 | 8 | 1 | 1 | 243.4 | 310.9 | -21.70 | -1.63 |
| | 1 | 3 | 3 | 4.431 | 4.499 | -1.51 | -1.29 | 1 | 1 | 2 | 2 | 3.236 | 3.115 | 3.89 | -1.89 |
| | 2 | 4 | 4 | 3.039 | 3.098 | -1.93 | -1.94 | 2 | 2 | 1 | 1 | 2.496 | 2.567 | -2.75 | -2.74 |
| | 3 | 5 | 5 | 2.492 | 2.559 | -2.59 | -2.58 | 3 | 3 | 1 | 1 | 2.391 | 2.476 | -3.45 | -3.45 |
| FIR | 4 | 6 | 6 | 2.420 | 2.503 | -3.30 | -3.30 | 4 | 4 | 1 | 1 | 2.067 | 2.133 | -3.09 | -3.09 |
| | 5 | 7 | 7 | 2.319 | 2.390 | -2.96 | -3.41 | 5 | 5 | 1 | 1 | 2.219 | 2.330 | -4.77 | -3.09 |
| | 6 | 8 | 8 | 2.193 | 2.250 | -2.51 | -4.09 | 6 | 6 | 1 | 1 | 2.153 | 2.340 | -7.99 | -2.58 |
| | 7 | 9 | 9 | 2.118 | 2.283 | -7.23 | -4.21 | 7 | 7 | 1 | 1 | 2.127 | 2.312 | -8.01 | -2.00 |
| | 8 | 10 | 10 | 2.017 | 2.044 | -1.36 | -4.37 | 8 | 8 | 1 | 1 | 2.061 | 2.108 | -2.26 | -1.42 |
| | 1 | 8 | 8 | 0.754 | 0.779 | -3.22 | -0.06 | 2 | 2 | 3 | 3 | 0.303 | 0.286 | 6.05 | -0.13 |
| | 2 | 9 | 9 | 0.441 | 0.441 | -0.29 | -0.29 | 3 | 3 | 2 | 2 | 0.209 | 0.209 | -0.1 | -0.70 |
| | 3 | 10 | 10 | 0.33 | 0.332 | -0.6 | -0.60 | 4 | 4 | 2 | 2 | 0.178 | 0.177 | 0.36 | -1.07 |
| Lm1 | 4 | 11 | 11 | 0.26 | 0.26 | -0.22 | -0.22 | 5 | 5 | 2 | 2 | 0.153 | 0.153 | -0.2 | -0.60 |
| | 5 | 12 | 12 | 0.239 | 0.229 | 4.17 | -0.25 | 6 | 6 | 1 | 1 | 0.149 | 0.163 | -8.33 | -0.42 |
| | 6 | 13 | 13 | 0.218 | 0.208 | 4.52 | -1.86 | 7 | 7 | 1 | 1 | 0.145 | 0.152 | -4.74 | -1.04 |
| | 7 | 14 | 14 | 0.198 | 0.19 | 4.52 | -0.30 | 8 | 8 | 1 | 1 | 0.137 | 0.163 | -15.97 | -0.38 |
| | 8 | 15 | 15 | 0.187 | 0.19 | -1.75 | -0.32 | 9 | 9 | 1 | 1 | 0.134 | 0.164 | -18.6 | -0.32 |
| | 1 | 10 | 10 | 5.712 | 5.367 | 6.42 | -0.30 | 8 | 8 | 1 | 1 | 6.371 | 5.411 | 17.75 | 0.44 |
| | 2 | 13 | 13 | 3.713 | 3.736 | -0.61 | -3.21 | 8 | 8 | 1 | 2 | 3.29 | 3.249 | 1.29 | 1.29 |
| | 3 | 17 | 17 | 3.359 | 3.389 | -0.87 | -0.87 | 12 | 12 | 1 | 1 | 3.85 | 3.884 | -0.89 | -0.89 |
| Jacobi | 4 | 20.5 | 21 | 3.095 | 3.125 | -0.95 | -3.31 | 16 | 16 | 1 | 1 | 4.695 | 4.642 | 1.13 | 1.13 |
| | 5 | 24 | 25 | 2.948 | 3.186 | -7.47 | -4.95 | 20 | 20 | 1 | 1 | 4.684 | 4.286 | 9.28 | 2.06 |
| | 6 | 27.5 | 29 | 2.972 | 3.139 | -5.34 | -5.70 | 24 | 24 | 1 | 1 | 4.821 | 4.931 | -2.23 | 0.43 |
| | 7 | 31 | 33 | 2.842 | 3.137 | -9.42 | -6.78 | 28 | 28 | 1 | 1 | 4.758 | 5.405 | -11.97 | 2.26 |
| | 8 | 34.5 | 37 | 2.854 | 3.177 | -10.17 | -7.16 | 32 | 32 | 1 | 1 | 4.863 | 5.912 | -17.74 | 2.21 |
| | 1 | 5 | 5 | 0.313 | 0.312 | 0.36 | -0.04 | 1 | 1 | 4 | 4 | 0.122 | 0.126 | -3.29 | 0.44 |
| | 2 | 6 | 6 | 0.188 | 0.188 | -0.1 | -0.10 | 2 | 2 | 2 | 2 | 0.092 | 0.092 | -0.19 | 1.29 |
| | 3 | 7 | 7 | 0.151 | 0.151 | -0.17 | -0.17 | 3 | 3 | 2 | 2 | 0.091 | 0.092 | -0.36 | -0.89 |
| Histogram | 4 | 8 | 8 | 0.126 | 0.126 | -0.15 | -0.15 | 4 | 4 | 1 | 1 | 0.102 | 0.102 | -0.23 | 1.13 |
| | 5 | 9 | 9 | 0.116 | 0.121 | -4.43 | -1.23 | 5 | 5 | 1 | 1 | 0.1 | 0.111 | -10.34 | -1.24 |
| | 6 | 10 | 10 | 0.107 | 0.113 | -5.09 | -1.56 | 6 | 6 | 1 | 1 | 0.097 | 0.109 | -11.18 | -1.56 |
| | 7 | 11 | 11 | 0.102 | 0.105 | -2.86 | -0.31 | 7 | 7 | 1 | 1 | 0.095 | 0.135 | -29.84 | -0.31 |
| | 8 | 12 | 12 | 0.097 | 0.105 | -7.57 | -0.20 | 8 | 8 | 1 | 1 | 0.092 | 0.12 | -23.07 | 2.21 |

**Table 1.** Accuracy of Performance Model.

| **Benchmark** | **Unrolling** | | | | **Unrolling + SWP** | | | |
|---|---|---|---|---|---|---|---|---|
| | $m_{\text{unrolled}}$ | $c_{\text{unrolled}}$ | **Max Err.** | **Min Err.** | $m_{\text{swp}}$ | $c_{\text{swp}}$ | **Max Err.** | **Min Err.** |
| Edge | 10,371 | 14,826 | 1.67% | 0.53% | 13,471 | 22,104 | 11.67% | 0.92% |
| MM (large) | 10,468 | 15,333 | 4.55% | 1.52% | 13,365 | 24,686 | 20.86% | 0.71% |
| FIR | 9,496 | 13,115 | 4.18% | 3.91% | 11,701 | 20,457 | 9.87% | 0.32% |
| Lm1 | 11,112 | 19,995 | 5.78% | 0.44% | 13,926 | 28,955 | 1.42% | 0.08% |
| Jacobi | 6,604 | 12,157 | 2.02% | 0.63% | 9,039 | 25,908 | 4.08% | 0.37% |
| Histogram | 3,973 | 5,676 | 13.50% | 2.25% | 3,714 | 1,6505 | 2.51% | 0.01% |

**Table 2.** Accuracy of Linear Regression.

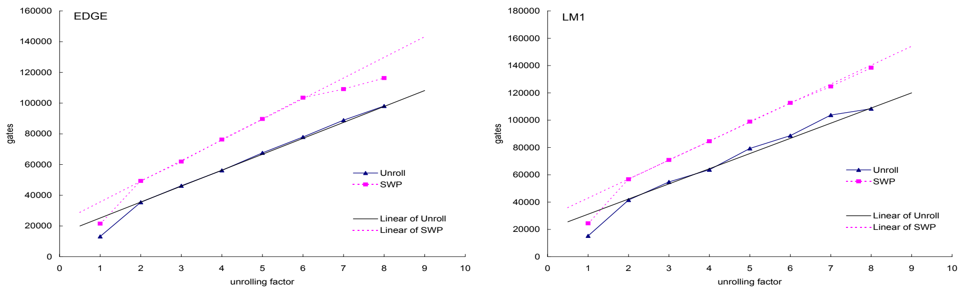

**Fig. 2.** Resource requirement for the benchmarks.

## 6 Conclusion

In this paper, we proposed a model that projects the data obtained from a small number of compilation and synthesis runs to obtain a global picture of the tradeoffs the designer faces in selecting between two loop optimizations, namely loop unrolling and software pipelining. The novelty of our approach is in the use of key parameters reported by the compiler to capture information about the machine configuration, data dependencies, and resource requirement patterns. This allowed us to obtain a very accurate model of the the cycle counts of the loops' execution. In the worst case, we are less than 5% off the actual cycle counts for larger loops.

The big challenge has been in modeling the two key parameters obtainable only after place and route, namely the circuit's realizable frequency and the resource consumption. For resource usage, we found good linear relations in the growth of resource consumption as unrolling increases especially within the realistic unroll factors that we studied.

Our approach is not very satisfying in modeling the frequency of software pipelined loops. In the worse case for software pipelined loop with high unroll numbers, we are can be off by 30%. Nonetheless, taken together as a whole, the average relative error in estimating $T_{\text{swp}}(u)$ is 8.4%. We would certainly like to improve this in future works.

Combining the resource model and the performance model, we have a methodology for deciding the optimal unroll factor as well as predict whether software pipelining will be beneficial given a certain resource constraint given by the user. We believe our model will reduce the time for design space exploration.

## References

1. Rau, B.R.: Iterative Modulo Scheduling. The International Journal of Parallel Processing **24** (1996)
2. Page, I., Luk, W.: Compiling OCCAM into FPGAs. In: Proceedings of the International Symposium on Field Programmable Logic (FPL). (1991)
3. Rinker, R., et al.: An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware. IEEE Transactions on VLSI Systems **9** (2001)
4. Goldstein, S.C., et al.: Piperench: A Reconfigurable Architecture and Compiler. IEEE Computer (2000)
5. Callahan, T., Hauser, J.R., Wawrzynek, J.: The Garp Architecture and C Compiler. IEEE Computer (2000)
6. Weinhardt, M.: Compilation and Pipeline Synthesis for Reconfigurable Architectures. In: Proceedings of the Reconfigurable Architecture Workshop (RAW). (1997)
7. Weinhardt, M., Luk, W.: Pipeline vectorization for reconfigurable systems. In: Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM). (1999)
8. Babb, J., et al.: Parallelizing Applications into Silicon. In: Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM). (1999)
9. Snider, G., Shackleford, B., Carter, R.J.: Attacking the Semantic Gap between Application Programming Languages and Configurable Hardware. In: Proceedings of ACM FPGA. (2001)
10. Jones, A., et al.: PACT HDL: A C Compiler Targeting ASICs and FPGAs with Power and Performance Optimizations. In: Proceedings of International Conference on Compilers, Architecture. and Synthesis for Embedded Systems (CASES). (2002)
11. Schreiber, R.: High-Level Synthesis of Nonprogrammable Hardware Accelerators. In: Proceedings of the IEEE International Conference on Application Specific Systems, Architectures, and Processors (ASAP). (2000)
12. Sivaraman, M., Aditya, S.: Cycle-time Aware Architecture Synthesis of Custom Hardware Accelerator. In: Proceedings of International Conference on Compilers, Architecture. and Synthesis for Embedded Systems (CASES). (2002)
13. Mentor Graphics Inc.: Mentor Graphics Monet User's Manual (release r42). (1999)
14. Derrien, S., Rajopadhye, S.: Loop Tiling for Reconfigurable Accelerators. In: Proceedings of the International Symposium on Field Programmable Logic (FPL). (2001)
15. So, B., Hall, M.W., Diniz, P.C.: A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. In: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI). (2002)
16. Kathail, V., Schlansker, M., Rau, B.: Hpl-pd architectural specifications: Version 1.1. Technical Report Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories (Revised 2000)
17. Trimaran Consortium: TRIMARAN: An Infrastructure for Research in Instruction Level Parallelism. (http://www.trimaran.org)
18. Celoxica Inc.: Handel-C. (http://www.celoxica.com/tech/handel-c/)
19. Electronic Industries Alliance: Electronic Design Interface Format. (http://www.edif.org)