Efficient Pipelining of Nested Loops: Unroll-and-Squash

by

Darin S. Petkov

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2001

Author...
Department of Electrical Engineering and Computer Science
December 15, 2000

Supervised by ....
Randolph E. Harr
Director of Research, Synopsys, Inc.
Thesis Supervisor

Certified by...
Saman P. Amarasinghe
Assistant Professor, MIT Laboratory for Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

Efficient Pipelining of Nested Loops:

Unroll-and-Squash

by

Darin S. Petkov

Submitted to the Department of Electrical Engineering and Computer Science
on December 20, 2000, in partial fulfillment of
the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The size and complexity of current custom VLSI have forced the use of high-level programming languages to describe hardware, and compiler and synthesis technology to map abstract designs into silicon. Many applications operating on large streaming data usually require a custom VLSI because of high performance or low power restrictions. Since the data processing is typically described by loop constructs in a high-level language, loops are the most critical portions of the hardware description and special techniques are developed to optimally synthesize them. In this thesis, we introduce a new method for mapping nested loops into hardware and pipelining them efficiently. The technique achieves fine-grain parallelism even on strong intra- and inter-iteration data-dependent inner loops and, by economically sharing resources, improves performance at the expense of a small amount of additional area. We implemented the transformation within the Nimble Compiler environment and evaluated its performance on several signal-processing benchmarks. The method achieves up to 2x increase in the area efficiency compared to the best known optimization techniques.

Thesis Supervisors:

Randolph E. Harr
Director of Research, Advanced Technology Group, Synopsys, Inc.

Saman P. Amarasinghe
Assistant Professor, MIT Laboratory for Computer Science

Acknowledgements

I want to thank my thesis and research advisor Saman Amarasinghe for supporting me throughout my time at MIT. His profound knowledge of compilers and computer architectures has made a major impact on my academic and professional development, and I appreciate his invaluable help and advice in many areas – from doing research to applying for a job.

This work would not exist without Randy Harr, my project supervisor at Synopsys. His ideas along with the numerous mind-provoking discussions form the backbone of this research. His stories and thorough answers to all my questions have greatly contributed to my understanding of the computer industry.

I have very much enjoyed working with my colleagues from the Nimble Compiler group at Synopsys – Yanbing Li, Jon Stockwood and Yumin Zhang. They provided me with prompt assistance, invaluable feedback, and lots of fun time. I would also like to thank the Power group at Synopsys for the two great internships that I had with them, and specifically Srinivas Raghvendra for his trust in my professional abilities.

I want to thank my family for always being there for me, on the other side of the phone line, 5,000 miles away, supporting my struggle through the hard years at MIT. I would also like to thank all my friends from the Bulgarian Club at MIT for the great party time and happy memories from my college years.

Finally, I thank Gergana for her friendship and love. She changed my life completely… for good.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Growing consumer market needs that require processing of large amounts of streaming data with a limited power or dollar budget have led to the development of increasingly complex embedded systems and application-specific integrated circuits (ASIC). As a result, high-level compilation and sophisticated state-of-the-art computer-aided design (CAD) tools that synthesize custom silicon from abstract hardware-description languages are used to automate and accelerate the intricate design process. These techniques not only eliminate the need of human intervention at every stage of the design cycle, but also raise the level of abstraction and bring the hardware design closer and closer to the system engineer.

Various studies show that loops are the most critical parts of many applications. For example, Table 1.1 demonstrates that several popular signal-processing algorithms spend, on average, 95% of the execution time in a few computation-intensive loops. Thus, since loops are the application performance bottleneck, the new generation of CAD tools needs to borrow many transformation and optimization methods from traditional compilers to efficiently synthesize hardware from high-level languages. A large body of work exists on translating software applications from common programming languages such as C/C++ and Fortran for optimal sequential or parallel execution on conventional microprocessors. These techniques include software pipelining [16][19] for exploiting loop parallelism within single processors and loop parallelization for multi-processors [14].

Benchmark	# loops	# loops > 1% time	Total % (> 1 % time)
Wavelet image compression	25	13	99%
EPIC encoding	132	13	92%
UNEPIC decoding	62	15	99%
Media Bench ADPCM	3	3	98%
MPEG-2 encoder	165	14	85%
Skipjack encryption	6	2	99%

Table 1.1: Program execution time in loops.

However, a direct application of these methods fails to generate efficient hardware since the design tradeoffs in software compilation to a microprocessor and in the process of circuit synthesis from a program are rather different. For instance, the number of extra operators (instructions) resulting from a particular software compiler transformation may not be critical as long as it increases the overall parallelism in a microprocessor. On the other hand, the amount of additional area coming from new or duplicated operators that the hardware synthesis produces may have a much bigger impact on the performance and cost of a custom VLSI (very large-scale integrated circuit) design. Furthermore, in contrast to traditional compilers, which are restrained by the paucity of registers in general-purpose processors and their limited capacity to transfer data between registers and memory, hardware synthesis algorithms usually have much more freedom in allocating registers and connecting them to memory. In addition to that, custom silicon provides a lot of flexibility in choosing the optimal delay of each operator versus its size and allows application-specific packing of different operations into a single operator to achieve better performance.

When an inner loop has no loop-carried dependencies across iterations, many techniques such as pipelining and unrolling will provide efficient and effective parallel performance for both microprocessors and custom VLSI. Unfortunately, a large number of loops in practical signal-processing applications have strong loop-carried data dependencies. Many cryptographic algorithms, such as unchained Skipjack and DES for example, have a nested loop structure where an outer loop traverses the data stream while the inner loop transforms each data block. Furthermore, the outer loop has no strong inter-iteration data-dependencies while the inner loop has both inter- and intra-iteration dependencies that prevent synthesis tools employing traditional compilation techniques from mapping and pipelining them efficiently.

This thesis introduces a new loop transformation that efficiently maps nested loops following this pattern into hardware. The technique, which we call *unroll-and-squash*, exploits the outer loop parallelism, concentrates more computation in the inner loop and improves the performance with little area increase by allocating the hardware resources without expensive multiplexing and complex routing. The algorithm was prototyped using the Nimble Compiler environment [1], and its performance was evaluated on several signal-processing benchmarks. Unroll-and-squash reaches performance comparable to the best applicable traditional loop transformations with 2 to 10 times less area.

The rest of this document is organized as follows. Chapter 2 provides several simple examples as well as one practical application that motivated this work. Chapter 3 gives a brief overview of the loop transformation and optimization theory including dependence analysis and some relevant traditional loop transformations. Chapter 4 presents the unroll-and-squash algorithm along with the requirements for the legality of the transformation. Chapter 5 discusses the implementation of the method within the Nimble Compiler framework, and, subsequently, Chapter 6 demonstrates the benchmark results obtained using the technique. The document concludes with a concise summary of the work related to unroll-and-squash and briefly states the contributions of this thesis.

Chapter 2

Motivation

```
for (i=0; i<M; i++) {
  a = data_in[i];
  for (j=0; j<N; j++) {
    b = f(a);
    a = g(b);
  }
  data_out[i] = a;
}
```

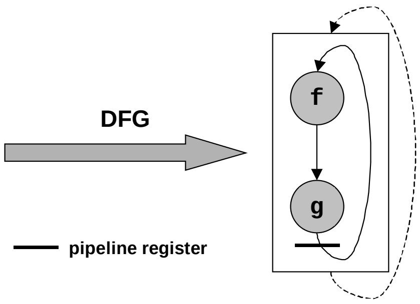

Figure 2.1: A simple example of a nested loop.

The importance and the application of the new technique can be demonstrated using the simple set of loops shown in Figure 2.1. Although it is trivial, this loop nest represents the common pattern that many digital signal-processing algorithms follow. The outer loop traverses M blocks of input data and writes out the result, while the inner loop runs the data through N rounds of two operators – `f` and `g`, each completing in 1 clock cycle. Little can be done to optimize this program considering only the inner loop. Because of the cycle in the inner loop, it cannot be pipelined, i. e., it is not possible to execute several inner loop iterations in parallel. Also, there is no instruction-level parallelism (ILP) in the inner loop basic block. The interval at which consecutive iterations are started is called the *initiation interval (II)*. As depicted in the *data-flow* *graph* (DFG), the minimum II of the inner loop is 2 cycles, and the total time for the loop nest is $2 \times M \times N$.

```
for (i=0; i<M; i+=2) {
  a1=data_in[i]; a2=data_in[i+1];
  for (j=0; j<N; j++) {
    b1 = f(a1); b2 = f(a2);
    a1 = g(b1); a2 = g(b2);
  }
  data_out[i]=a1; data_out[i+1]=a2;
}
```

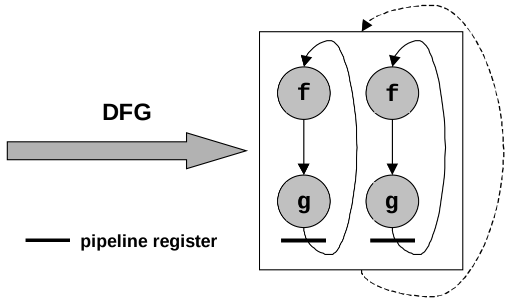

Figure 2.2: A simple example: unroll-and-jam by 2.

Traditional loop optimizations such as loop unrolling, flattening, permutation and pipelining [29] fail to exploit the parallelism and improve the performance for this set of loops. One successful approach in this case is the application of *unroll-and-jam* (Figure 2.2), which unrolls the outer loop but fuses the resulting sequential inner loops to maintain a single inner loop [28], explained further in Chapter 3. After applying unroll-and-jam with a factor of 2 (assuming that M is even), the resulting inner loop has 4 operators (twice the original number). Although this transformation does not decrease the minimum II of the inner loop because the data-dependency cycle still exists, the ability to execute several operators in parallel has the potential to speed up the program. The II is 2 but the total execution time is half the original since the outer loop iteration count is halved – $2 \times (M/2) \times N = M \times N$. Thus, unroll-and-jam doubles the performance of the application at the expense of a doubled operator count.

A more efficient way to improve the performance of this sample set of loops is by applying the unroll-and-squash technique introduced in this thesis, which decreases the overall execution time of the original program without a significant amount of additional area. This transformation, similarly to unroll-and-jam, unrolls the outer loop but maintains a single inner loop that executes the consecutive outer loop iterations in parallel. However, the data sets of the different outer loop iterations run through the inner loop operators in a round-robin manner, which allows the parallel execution of the operators and a lower II. Moreover, the transformation adds to the hardware implementation of the inner loop only registers and, since the original operator count remains unchanged, the design area stays approximately the same.

```
for (i=0; i<M; i+=2) {
  a1=data_in[i]; a2=data_in[i+1];
  b1 = f(a1);
  for (j=0; j<2*N-1; j++) {
    b2 = f(a2); a1 = g(b1);
    a2 = a1; b1 = b2;
  }
  a1 = g(b1);
  data_out[i]=a2; data_out[i+1]=a1;
}
```

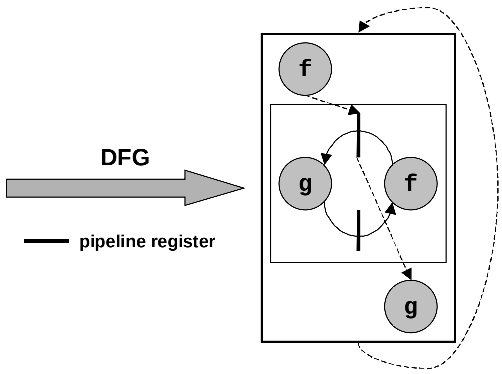

Figure 2.3: A simple example: unroll-and-squash by 2.

The application of unroll-and-squash on the sample loop nest by a factor of 2, illustrated in Figure 2.3, is similar to unroll-and-jam with respect to the transformation of the outer loop – the outer loop iteration count is halved, and 2 outer loop iterations are processed in parallel. However, the operator count in the inner loop remains the same as in the original program – 2. By adding variable shifting/rotating statements, which translate into register moves in hardware, and pulling appropriate prolog and epilog out of the inner loop to fill and flush the pipeline, the transformation can be correctly expressed in software. One should note that these extra source code statements might not be necessary if a pure hardware implementation is pursued. Since the final II is 1, the total execution time of the loop nest is $1\times(M/2)\times(2\times N)=M\times N$. Thus, unroll-and-squash doubles the performance without paying the additional cost of extra operators.

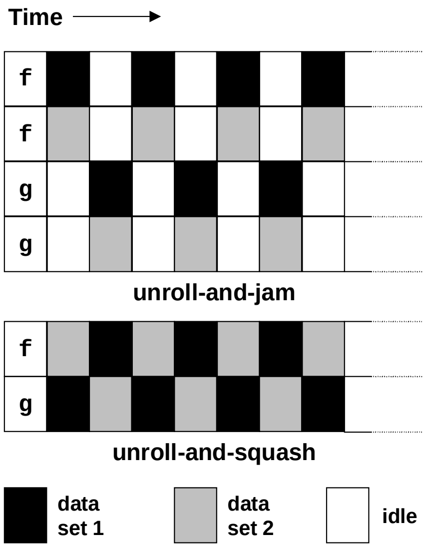

Figure 2.4: Operator usage.

Figure 2.4 shows the operator usage over time in the unroll-and-jammed and unroll-and-squashed versions of the program (it omits the prolog and the epilog necessary for unroll-and-squash). Besides the fact that unroll-and-squash makes better use of the existing operators than unroll-and-jam by filling all available idle time slots, another important observation is that it may be possible to combine both techniques simultaneously. Unroll-and-jam can be applied with an unroll factor that matches the desired or available amount of operators, and then unroll-and-squash can be used to further improve the performance and achieve better operator utilization. For example, after applying unroll-and-jam by a factor of 2 to the sample loop nest, which doubles both the performance and the operator count, a subsequent unroll-and-squash transformation by a factor of 2 further speeds up the program without a significant amount of extra area. The execution time is $1\times(M/4)\times(2\times N)=M\times N/2$ and the inner loop operator count is 4.

That is, the combined application of the two transformations quadruples the performance but only doubles the area. It is important to notice that the sole use of unroll-and-squash by a factor of 4 in this case will be less beneficial for the execution time.

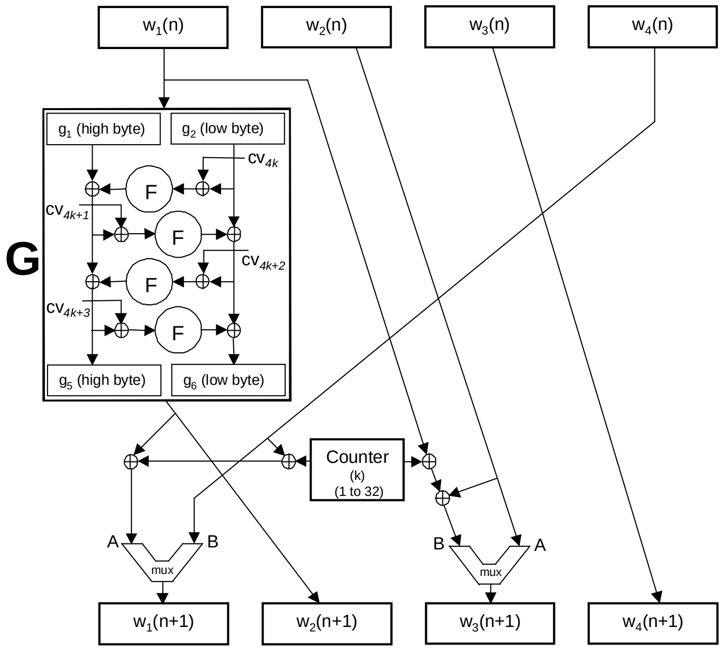

Figure 2.5: Skipjack cryptographic algorithm.

A good example of a real-world application of unroll-and-squash is the Skipjack cryptographic algorithm, declassified and released in 1998 (Figure 2.5). This crypto-algorithm encrypts 8-byte data blocks by running them through 32 rounds of 4 table-lookups (*F*) combined with key-lookups (*cv*), a number of logical operations and input selection. The *F*-lookups form a long cycle that prevents the encryption loop from being efficiently pipelined. Again, little can be done by optimizing the inner loop in isolation but, as with the simple example in the previous section, proper application of unroll-and-squash (separately or together with unroll-and-jam) on the outer, data-traversal loop can boost the performance significantly at a low extra area cost.

Chapter 3

Loop Transformation Theory Overview

This chapter gives a brief overview of the loop transformation theory including data dependence analysis and several examples of traditional loop transformations relevant to the unroll-and-squash method. More comprehensive presentations of the loop transformation theory can be found in [27], [29] and [30]. Other applicable compiler analysis and optimization techniques are discussed in Chapter 4.

3.1 Iteration Space Graph

A FOR style loop nest of depth n can be represented as an *iteration space graph* with axes corresponding to the different loops in the loop nest (Figure 3.1). The axes are labeled with the related index variables and limited by the loop iteration bounds. Each iteration is represented as a node in the graph and identified by its index vector $\vec{p} = (p_1, p_2, \ldots, p_n)$, where p_i is the value of the ith loop index in the nest, counting from the outermost to the innermost loop. Assuming positive loop steps, we can define that iteration index vector $\vec{p}$ is *lexicographically greater than* $\vec{q}$, denoted by $\vec{p} \succ \vec{q}$, if and only if $p_1 > q_1$ or both $p_1 = q_1$ and $(p_2, \ldots, p_n) \succ (q_2, \ldots, q_n)$. Additionally, $\vec{p} \succcurlyeq \vec{q}$ if and only if either $\vec{p} \succ \vec{q}$, or $\vec{p} = \vec{q}$. In general, iteration $\vec{p}$ will execute after iteration $\vec{q}$ if and only if $\vec{p} \succ \vec{q}$. The execution order can be represented as arcs between the nodes in the iteration space graph specifying the *iteration-space traversal*. The *data dependences*, i. e., the ordering constraints between the iteration nodes, determine alternative *valid* execution orderings of the nodes that are semantically equivalent to the lexicographic node execution.

```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    S(i,j);
```

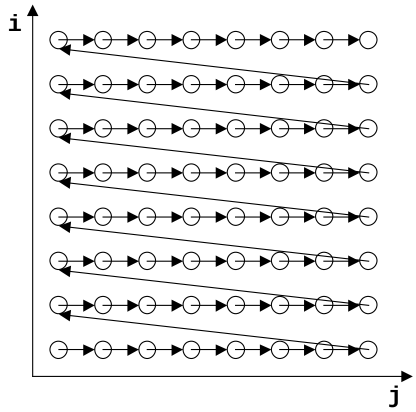

Figure 3.1: Iteration-space graph.

The iteration execution ordering can be extended to single loop operations using the "$\triangleright$" notation. Given two operations $S_1[\vec{p}]$ and $S_2[\vec{q}]$, where $\vec{p}$ and $\vec{q}$ are the iterations containing S_1 and S_2 respectively, $S_1[\vec{p}] \triangleright S_2[\vec{q}]$ means that $S_1[\vec{p}]$ is executed after $S_2[\vec{q}]$. In general, $S_1[\vec{p}] \triangleright S_2[\vec{q}]$ if and only if either S_1 follows S_2 in the operation sequence and $\vec{p} \succcurlyeq \vec{q}$, or S_1 is the same operation as or precedes S_2 and $\vec{p} \succ \vec{q}$. Similarly to the iteration-space traversal, the operation execution ordering can also be displayed graphically.

3.2 Data Dependence

Given two memory accesses to the same memory location $S_1[\vec{p}]$ and $S_2[\vec{q}]$ such that $S_1[\vec{p}] \triangleright S_2[\vec{q}]$, there is said to be data dependence between the two operations and, consequently, the two iterations. Distance and dependence vectors are used to describe such loop-based data dependences. A *distance vector* for an *n*-loop nest is an *n*-dimensional vector $\vec{\delta} = (\delta_1, \ldots, \delta_n)$ such that the iteration with index vector $\vec{p} + \vec{\delta} = (p_1 + \delta_1, \ldots, p_n + \delta_n)$ depends on the one with index vector $\vec{p}$. If there is data dependence between $S_1[\vec{p}]$ and $S_2[\vec{q}]$, the distance vector is $\vec{\delta} = \vec{p} - \vec{q}$. A *dependence vector* for an *n*-loop nest is an *n*-dimensional vector $\vec{d} = ([d_1^-, d_1^+], \ldots, [d_n^-, d_n^+])$ that summarizes a set of distance vectors called its *distance vector set*:

$$DV(\vec{d}) = \{(\delta_1, \ldots, \delta_n) \mid d_i^- \le \delta_i \le d_i^+\}$$

Note that a dependence distance of $(0, \ldots, 0)$ has no effect on loop transformations that keep the order of individual operations and statements unchanged. Finally, a dependence may be *loop-independent* (that is, independent of the enclosing loops) or *loop-carried* (dependence due to the surrounding loops). Methods to determine loop data dependences include the extended GCD test, the strong and weak single index variable tests, the Delta test, the Acyclic test and others [15].

3.3 Tiling

Tiling is a loop transformation that increases the depth of a loop nest and rearranges the iteration-space traversal, often used to exploit data locality (Figure 3.2).

Given an *n*-loop nest, tiling may convert it to anywhere from *(n+1)*- to *2n*-deep loop nest. Tiling a single loop replaces it by a pair of loops – the inner one has a step equal to that of the original loop, and the outer one has a step equal to *ub-lb+1*, where *lb* and *ub* are, respectively, the lower and upper bounds of the inner loop. The number of iterations of the inner (tile) loop is called the *tile size*. In general, tiling a loop nest is legal if and only if the loops in the loop nest are fully permutable. A proof of this statement can be found in [30].

```
for (ii=0; ii<M; ii+=Si)
  for (jj=0; jj<N; jj+=Sj)
    for (i=ii; i<ii+Si-1; i++)
      for (j=jj; j<jj+Sj-1; j++)
        S(i,j);
```

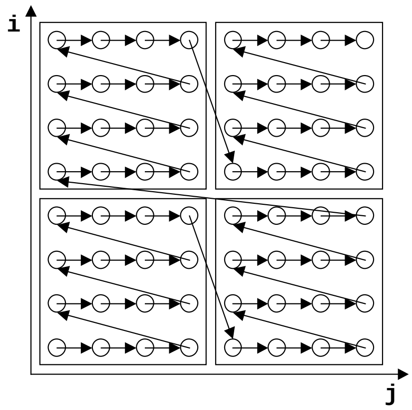

Figure 3.2: Tiling the sample loop in Figure 3.1.

3.4 Unroll-and-Jam

Unroll-and-jam, demonstrated in Figure 3.3, is a sequence of two loop transformations, unrolling and fusion, applied to a 2-loop nest. *Loop unrolling* replaces a loop body by several copies of the body, each operating on a consecutive iteration. The number of copies of the loop body is called the *unroll factor*. Unrolling without additional operations is legal as long as the loop iteration count is a multiple of the unroll factor. *Loop fusion* is a loop transformation that takes two adjacent loops with the same iteration-space graphs and merges their bodies into a single loop. Fusion can be applied if the loops have the same bounds and there are no operations in the second loop dependent on operations in the first one. Finally, *unroll-and-jam* can be applied to a set of 2 nested loops by unrolling the outer loop and fusing the resulting sequential inner loops. Unroll-and-jam can be used as long as the outer loop unrolling and the subsequent fusion are legal. It may improve performance by concentrating more parallel computation in the inner loop, by exploiting data locality and by eliminating loop setup costs. However, it increases the amount of operations in the inner loop proportionally to the unroll factor.

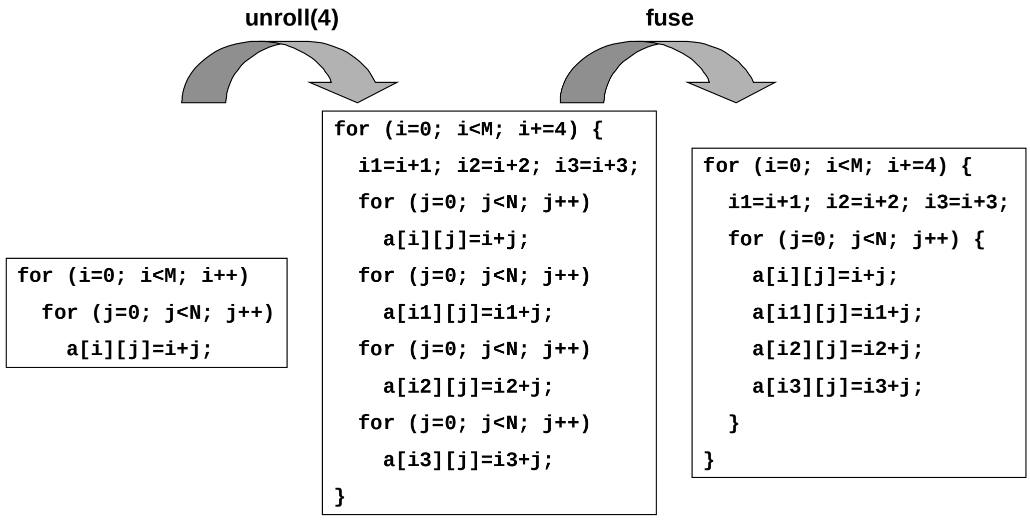

Figure 3.3: Unroll-and-jam by a factor of 4.

One should also note that unroll-and-jam can be represented as an alternative sequence of loop transformations – tiling the outer loop with a tile size equal to the unroll-and-jam factor, and full tile loop unrolling. This approach signifies the fact the unroll-and-jam changes the iteration space traversal order, and data dependence analysis should be employed to verify the legality of the transformation.

3.5 Pipelining

One of the most important and effective techniques for exploiting parallelism in loops is *loop pipelining* (software or hardware), illustrated in Figure 3.4. Let $\{fg\}^{n}$ be a loop where `f` and `g` denote the operators in the loop body and `n` is the iteration count. Pipelining relies on the fact that this loop is equivalent to $f\{gf\}^{n-1}g$ and improves performance by overlapping the execution of different iterations. The operators of the loop executed before the loop body after the transformation (`f`) form the loop *prolog*, the operators executed after the body (`g`) are the loop *epilog*, and the interval at which iterations are started is the *initiation interval (II)*. The goal of pipelining is to achieve the minimum possible II, which is hardware resource or data dependence constrained [16]. Combined with other loop transformations to eliminate the data dependences and enlarge the basic blocks, such as modulo variable expansion and loop unrolling and fusion, loop pipelining becomes a powerful method for exploiting the parallelism inherent to loops.

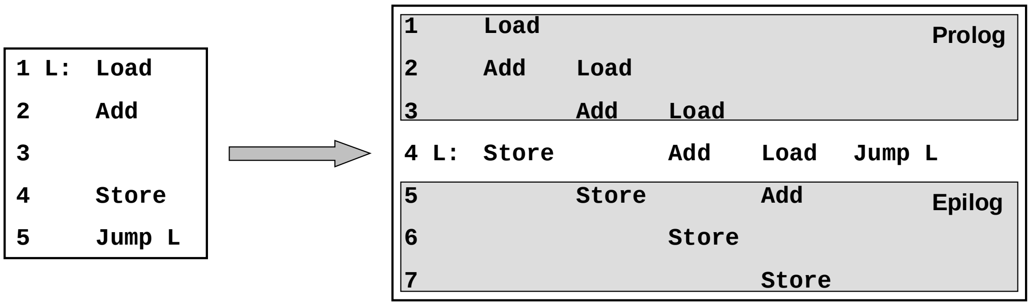

Figure 3.4: Loop pipelining in software.

Chapter 4

Unroll-and-Squash

The unroll-and-squash transformation optimizes the performance of 2-loop nests by executing multiple outer loop iterations in parallel. The inner loop operators cycle through the separate outer loop data sets, which allows them to work simultaneously. By doing efficient resource sharing, this technique reduces the total execution time without increasing the operator count. This chapter assumes that unroll-and-squash is applied to a nested loop pair where the outer loop iteration count is *M*, the inner loop iteration count is *N*, and the unroll factor is *DS* (*Data Sets*).

4.1 Requirements

This section outlines the general control-flow and data-dependency requirements that must hold for the proposed transformation to be applied to an inner-outer loop pair. In the next section, we show how some of these conditions can be relaxed by using various code analysis and transformation techniques such as induction variable identification, variable privatization, and others.

Unroll-and-squash can be applied to any set of 2 nested loops that can be successfully unroll-and-jammed [28]. For a given unroll factor *DS*, it is necessary that the outer loop can be tiled in blocks of *DS* iterations, and that the iterations in each block be parallel. The inner loop should comprise a single basic block and have a constant iteration count across the different outer loop iterations. The latter condition also implies that the control-flow always passes through the inner loop.

4.2 Compiler Analysis and Optimization Techniques

A number of traditional compiler analysis, transformation and optimization techniques can be used to determine whether a particular loop nest follows the requirements, to convert the loop nest to one that conforms with them, or to increase the efficiency of unroll-and-squash. First of all, most standard compiler optimizations that speed up the code or eliminate unused portions of it can be applied before unroll-and-squash. These include constant propagation and folding, copy propagation, dead-code and unreachable-code elimination, algebraic simplification, strength-reduction to use smaller and faster operators in the inner loop, and loop invariant code motion. Scalarization may be used to reduce the number of memory references in the inner loop and replace them with register-to-register moves. Although very useful, these optimizations can rarely enlarge the set of loops that unroll-and-squash can be applied to.

One way to eliminate conditional statements in the inner loop and make it a single basic block (one of the restrictions) is to transform them to equivalent logical and arithmetic expressions (if-conversion). Another alternative is to use code hoisting to move the conditional statements out of the inner-outer loop pair, if possible.

In order for the outer loop to be tiled in blocks of DS iterations, its iteration count M should be a multiple of DS. If this condition does not hold, loop peeling may be used, that is, *M mod DS* iterations of the outer loop may be executed independently from the remaining *M-(M mod DS)*.

The data-dependency requirement, i. e., the condition that the tiled iterations of the outer loop should be parallel, is much more difficult to determine or overcome. Moreover, if the outer loop data dependency is an innate part of the algorithm that the loop nest implements, it is usually impossible to apply unroll-and-squash. One approach to eliminate some of the scalar variable data dependencies in the outer loops is by induction variable identification – it can be used to convert all induction variable definitions in the outer loop to expressions of a single index variable. Another method is modulo variable expansion, which replaces a variable with several separate variables corresponding to different iterations and combines them at the end. If the loops contain array references, dependence analysis [27] may be employed to determine the applicability of the technique and array privatization may be used to better exploit the parallelism. Finally, pointer analysis and other relevant techniques (such as converting pointer to array accesses) may be utilized to determine whether code with pointer-based memory accesses can be parallelized.

The use of dependence analysis is summarized below. Let A_1 and A_2 be two different memory accesses inside the inner-outer loop pair. If the accesses are memory loads, they are independent for the purposes of the technique and, therefore, we assume that at least one of them is a memory store. Without losing generality, we can also assume that the outer loop is not enclosed by another loop. The dependence vector is defined as follows:

$\vec{d}(A_1, A_2) = ([d_1^-, d_1^+])$, if neither A_1, nor A_2 belongs to the inner loop, or

$\vec{d}(A_1, A_2) = ([d_1^-, d_1^+], [d_2^-, d_2^+])$, if either A_1, or A_2 belongs to the inner loop.

There are 3 cases for $[d_1^-, d_1^+]$ that need to be considered in order to determine whether the transformation can be applied to the particular inner-outer loop pair:

Case 1: $[d_1^-, d_1^+] = [0,0]$. If the dependence distance is 0 then the dependence is iteration-independent and the loop transformation will not introduce any data hazards – the unrolled memory accesses will be independent.

Case 2: $[d_1^-, d_1^+] \cap [-DS+1, DS-1] = \varnothing$. If the intersection between the outer loop dependence range and the data set range is empty, unroll-and-squash will not create any data hazards – any dependent accesses will be executed in different outer loop iterations.

Case 3: $[d_1^-, d_1^+] \neq [0,0]$ and $[d_1^-, d_1^+] \cap [-DS+1, DS-1] \neq \varnothing$. If the dependence distance is non-zero and the intersection between the outer loop dependence range and the data set range is non-empty, unroll-and-squash may reorder and execute the memory accesses incorrectly and introduce data hazards.

4.3 Transformation

Once it is determined that a particular loop pair can be unroll-and-squashed by an unroll factor *DS*, it is necessary to efficiently assign the functional elements in the inner loop to separate pipeline stages, and apply the corresponding transformation to the software representation of the loop. Although it is possible to have a pure hardware implementation of the inner loop (without a prolog and an epilog in software), the outer loop still needs to be unrolled and have a proper variable assignment. The sequence of basic steps that are used to apply unroll-and-squash to a loop nest are presented below:

```
for (i=0; i<M; i++) {
  a = in[i];
  for (j=0; j<N; j++) {
    b = a + i;
    c = b - j;
    a = (c & 15) * k;
  }
  out[i] = a;
}
```

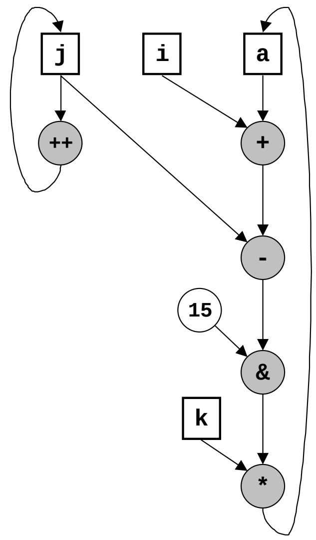

Figure 4.1: Unroll-and-squash – building the DFG.

- Build the DFG of the inner loop (Figure 4.1). Live variables are stored in registers at the top of the graph.
- Transform live variables that are used in the inner loop but defined in the outer loop (i. e., registers that have no incoming edges) into cycles (output edges from the register back to itself).
- "Stretch" the cycles in the graph so that the backedges start from the bottom and go all the way to the registers at the top.
- Pipeline the resulting DFG ignoring the backedges (Figure 4.2) producing exactly *DS* pipeline stages. Empty stages may be added or pipeline registers may be removed to adjust the stage count to *DS*.

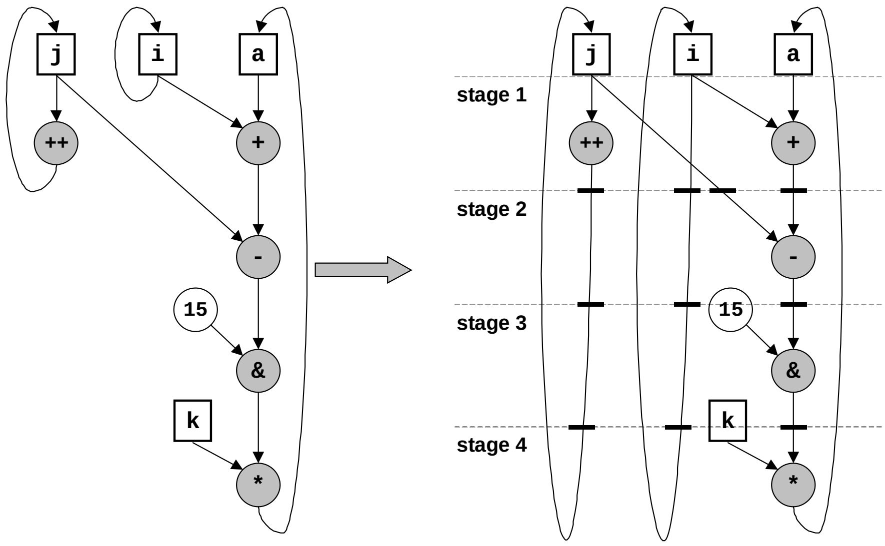

Figure 4.2: Stretching cycles and pipelining.

- Perform variable expansion – expand each variable in the inner/outer loop nest to *DS* versions. Some of the resulting variables may not actually be used later.
- Unroll the outer loop basic blocks (this includes the basic blocks that dominate and post-dominate the inner loop).
- Generate prolog and epilog code to fill and flush the pipeline (unless the inner loop is implemented purely in hardware).
- Assign proper variable versions in the inner loop. Note that some new (delay) variables may be needed to handle expressions split across pipeline registers.
- Add variable shifting/rotation to the inner loop. Note that reverse shifting/rotation may be required in the epilog or, alternatively, a proper assignment of variable versions.

The outer loop data sets pass through the pipeline stages in a round-robin manner. All live variables should be saved to and restored from the appropriate hardware registers before and after execution, respectively.

4.4 Algorithm Analysis

The described loop transformation decreases the number of outer loop iterations from M to M/DS. A software implementation will increase the inner loop iteration count from N to $DS \times N-(DS-1)$ and execute some of the inner loop statements in the prolog and the epilog in the outer loop. The total iteration count of the loop nest stays approximately the same as the original – $M \times N$.

There are several factors that need to be considered in order to determine the optimal unroll factor *DS*. One of the main barriers to performance increase is the maximum number of pipeline stages that the inner loop can be efficiently divided into. In a software implementation of the technique, this number is limited by the operator count in the critical path in the DFG or may be smaller if different operator latencies are taken into account. A pure hardware implementation bounds the stage count to the delay of the critical path divided by the clock period. The pipeline stage count determines the number of outer loop iterations that can be executed in parallel and, in general, the more data sets that are processed in parallel the better the performance. Certainly, the calculation of the unroll factor *DS* should be made in accordance to the outer loop iteration count (loop peeling may be required) and the data dependency analysis discussed in the previous section (larger *DS* may eliminate the parallelism).

Another important factor for determining the unroll factor *DS* is the extra area and, consequently, extra power that comes with large values of *DS*. Unroll-and-squash adds only pipeline registers to the existing operators and data feeds between them and, because of the cycle stretching, most of them can be efficiently packed in groups to form a single shift register. This optimization may decrease the impact of the transformation on the area and the power of the design, as well as make routing easier – no multiplexors are added, in contrast to traditional hardware synthesis techniques. In comparison with unroll-and-jam by the same unroll factor, unroll-and-squash results in less area since the operators are not duplicated. The tradeoff between speed, area and power is further illustrated in the benchmark report (Chapter 6).

Chapter 5

Implementation

Recently, there has been an increased interest in hardware/software co-design and co-synthesis both in the academia and in the industry. Most hardware/software compilation systems focus on the functional partitioning of designs amongst ASIC (hardware) and CPU (software) components [5][6][7]. In addition to using traditional behavioral synthesis languages such as Verilog and VHDL, synthesis from software application languages such as C/C++ or Java is also gaining popularity. Some of the systems that synthesize subsets of C/C++ or C-based languages include HardwareC [21], SystemC [22], and Esterel C [23]. DeepC, a compiler for a variation of the RAW parallel architecture presented in [2], allows sequential C or Fortran programs to be compiled directly into custom silicon or reconfigurable architectures. Some other novel hardware synthesis systems compile Java [24], Matlab [26] and term-rewriting systems [25]. In summary, the work in this field clearly suggests that future CAD tools will synthesize hardware designs from higher levels of abstraction. Some efforts in the last few years have been concentrated on automatic compilation and partitioning to reconfigurable architectures [8][9][10]. Callahan and Wawrzynek [3] developed a compiler for the Berkeley GARP architecture [4] which takes C programs and compiles them to a CPU and FPGA. The Nimble Compiler environment [1] extracts hardware kernels (inner loops that take most of the execution time) from C applications to accelerate on a reconfigurable co-processor. This system was used to develop and evaluate the loop optimization technique presented in this thesis.

5.1 Target Architecture

Figure 5.1 demonstrates an abstract model of the new class of architectures that the Nimble Compiler targets. The Agile hardware architecture couples a general purpose CPU with a dynamically reconfigurable coprocessor. Communication channels connect the CPU, the datapath, and the memory hierarchy. The CPU can be used to implement and execute control-intensive routines and system I/O, while the datapath provides a large set of configurable operators, registers and interconnects, allowing acceleration of computation-intensive parts of an application by flexible exploitation of ILP.

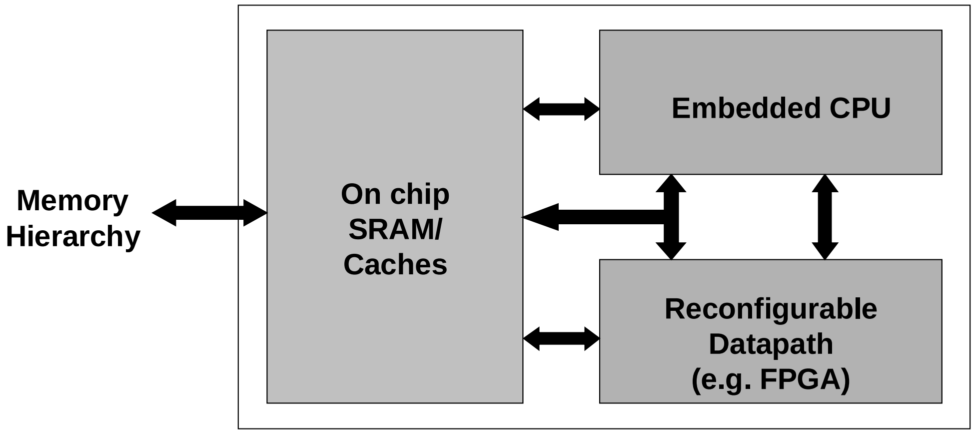

Figure 5.1: The target architecture – Agile hardware.

This abstract hardware model describes a broad range of possible architectural implementations. The Nimble Compiler is retargettable, and can be parameterized to target a specific platform described by an Architecture Description Language. The target platforms that the Nimble Compiler currently supports include the GARP architecture, the ACE2 card and the ACEV platform. Berkeley's GARP is a single-chip architecture with a MIPS 4000 CPU, a reconfigurable array of 24 by 32 CLBs, on-chip data and instruction caches, and a 4-level configuration cache [4]. The TSI Telsys ACE2 card is a board-level platform and consists of a μSparc CPU and Xilinx 4085 FPGAs [13]. The ACEV hardware prototype combines a TSI Telsys ACE card [12] with a μSparc CPU, and a PCI Mezzanine card [11], containing a Xilinx Virtex XCV 1000 FPGA. In the ACE card configurations, a fixed wrapper is defined in the FPGA to provide support resources to turn it into a configurable datapath coprocessor. The wrapper includes the CPU interface, memory interface, local memory optimization structures, and a controller.

5.2 The Nimble Compiler

The Nimble Compiler (Figure 5.2) extracts the computation-intensive inner loops (kernels) from C applications, and synthesizes them into hardware. The front-end, built using the SUIF compiler framework [14], profiles the program to obtain a full basic block execution trace along with the loops that take most of the execution time. It also applies various hardware-oriented loop transformations to concentrate as much of the execution time in as few kernels as possible, and generate multiple different versions of the same loop. Some relevant transformations include loop unrolling, fusion and packing, distribution, flattening, pipelining, function inlining, branch trimming, and others. A kernel selection pass chooses which kernel versions to implement in hardware based on the profiling data, a feasibility analysis, and a quick synthesis step. The back-end datapath synthesis tool takes the kernels (described as DFG's) and generates the corresponding FPGA bit streams that are subsequently combined with the rest of the C source code by an embedded compiler to produce the final executable binary.

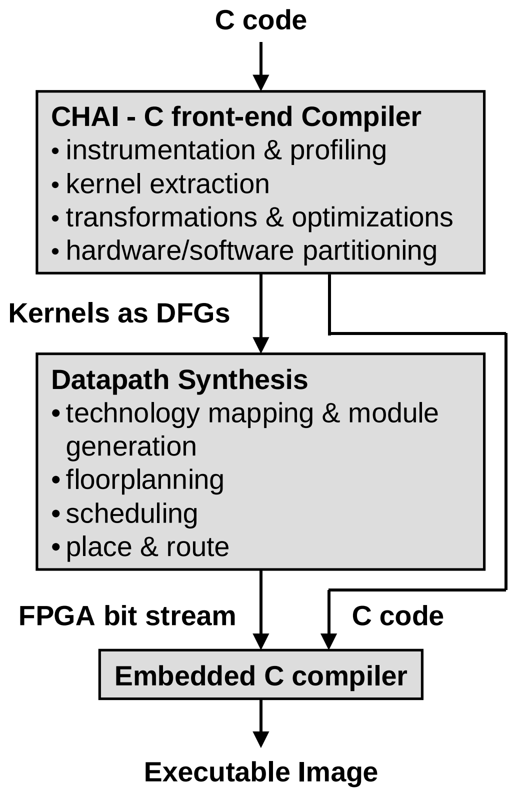

Figure 5.2: Nimble Compiler flow.

Unroll-and-squash is one of the loop transformations that the Nimble Compiler considers before kernel selection is performed. This newly discovered optimization benefits the Nimble Compiler environment in a variety of ways. First of all, outer loop unrolling concentrates more of the execution time in the inner loop and decreases the amount of transitions between the CPU and the reconfigurable datapath. In addition, this transformation does not increase the operator count and, assuming efficient implementation of the register shifts and rotation, the FPGA area is used optimally. Finally, unroll-and-squash pipelines loops with strong intra- and inter-iteration data dependencies and can be easily combined with other compiler transformations and synthesis optimizations.

5.3 Implementation Details

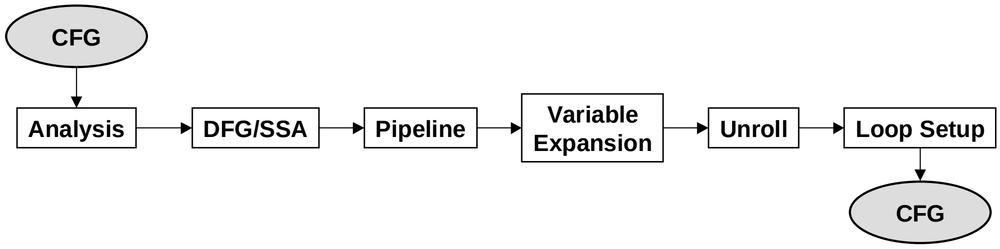

Figure 5.3: Unroll-and-squash implementation steps.

The unroll-and-squash transformation pass, depicted in Figure 5.3, was implemented in C++ within the Nimble Compiler framework. The module reads in a control-flow representation of the program using MachSUIF (an extension to SUIF for machine-dependent optimizations [31]) along with the loop, data dependence and liveness information, and finds the loop nests to be transformed, identified by user annotations. In the analysis step, the module checks the necessary control-flow and data dependency requirements. After determining the legality of the transformation, it builds a data-flow graph (DFG) for the inner loop instructions. The live variables in the DFG are represented by registers, and loop-carried dependences result in DFG backedges. While the DFG is built, the inner loop code is converted into *static single-assignment* (SSA) form, so that each variable is defined only once in the inner loop body. The pipeline step inserts pipeline registers in the DFG using the user-specified unroll factor *DS* and machine-dependent operator delays. It ignores the DFG backedges. Single expressions split by pipeline registers are transformed using temporary delay variables corresponding to the registers.

The subsequent steps express the unroll-and-squash transformation in software. First, variables are expanded into *DS* versions, and the outer loop basic blocks are unrolled. This involves assigning proper variable versions in the separate inner loop pipeline stages, as well as in the outer loop basic blocks corresponding to the different outer loop iterations. Also, variable shifting and rotation is added at the beginning of the inner loop. Then, a prolog to fill and an epilog to flush the inner loop pipeline are generated. These code transformation steps result in a modified program that can be correctly compiled and executed in software but may be much more efficiently mapped into hardware.

5.4 Front-end vs. Back-end Implementation

The unroll-and-squash transformation can be implemented either in the front-end, or the back-end of a hardware synthesis tool. A front-end implementation allows simple software representation of the transformed code and, specifically for the Nimble Compiler environment, permits an easy exploration of alternative optimizations. The key benefit of this approach is that it is flexible and permits a straightforward software-only compilation of the program.

The main disadvantage of implementing the technique in the front-end is the weak connection between the transformation and the actual hardware representation. One of the problems, for example, is that a software implementation in the front-end obstructs intra-operator pipelining because it manages whole operators. For benchmarking purposes, we modeled some operators such as floating point arithmetic to allow deeper pipelining. Another problem for the specific hardware target is that the back-end synthesis tool can pack different operators into a single row. Since the front-end has little knowledge about the possible mappings, it may actually pipeline the data-flow graph in a way that makes the performance worse in terms of both speed and area. A more sophisticated approach would integrate the unroll-and-squash transformation with the back-end and differentiate between the software transformation and the actual hardware representation.

Chapter 6

Experimental Results

We compared the performance of unroll-and-squash on the main computational kernels of several digital signal-processing benchmarks to the original loops, pipelined original loops, and pipelined unroll-and-jammed loops. The collected data shows that unroll-and-squash is an effective way to speed up such applications at a relatively low area cost and suggests that this is a valuable compiler and hardware synthesis technique in general.

6.1 Target Architecture Assumptions

The benchmarks were compiled using the Nimble Compiler with the ACEV target platform. Two memory references per clock cycle were allowed, and no cache misses were assumed. The latter assumption is not too restrictive for comparison purposes because the different transformed versions have similar memory access patterns. Furthermore, a couple of the benchmarks were specially optimized for a hardware implementation and had no memory references at all.

6.2 Benchmarks

Benchmark	Description
Skipjack-mem	Skipjack cryptographic algorithm: encryption, software implementation with memory references
Skipjack-hw	Skipjack cryptographic algorithm: encryption, software implementation optimized for hardware without memory references
DES-mem	DES cryptographic algorithm: encryption, SBOX implemented in software with memory references
DES-hw	DES cryptographic algorithm: encryption, SBOX implemented in hardware without memory references
IIR	4-cascaded IIR biquad filter processing 64 points

Table 6.1: Benchmark description.

The benchmark suit consists of two cryptographic algorithms (unchained Skipjack and DES) and a filter (IIR) described in Table 6.1. Two different versions of Skipjack and DES are used. Skipjack-mem and DES-mem are regular software implementations of the corresponding crypto-algorithms with memory references. Skipjack-hw and DES-hw are versions specifically optimized for a hardware implementation – they use local ROM for memory lookups and domain generators for particular bit-level operations. Finally, IIR is a floating-point filter implemented on the target platform by modeling pipelinable floating-point arithmetic operations.

6.3 Results and Analysis

Table 6.2 presents the raw data collected through our experiments. It compares ten different versions of each benchmark – an original, non-pipelined version, a pipelined version, unroll-and-squashed versions by factors of 2, 4, 8 and 16, and, finally, pipelined unroll-and-jammed versions by factors of 2, 4, 8 and 16. The table shows the initiation interval in clock cycles, the area of the designs in rows and the register count. One should note that if the initial loop pair iteration count is $M \times N$, after unroll-and-jam by a factor DS it becomes $M \times N/DS$.

Benchmark		original	pipelined	squash(2)	squash(4)	squash(8)	squash(16)	jam(2)	jam(4)	jam(8)	jam(16)
Skipjack-mem	II (cycles)	22	21	12	9	8	7	23	28	38	70
	Area (rows)	49	57	62	91	143	256	111	219	435	867
	Registers (count)	6	13	18	44	92	197	25	49	97	193
Skipjack-hw	II (cycles)	19	19	11	7	4	3	19	19	19	19
	Area (rows)	41	41	56	86	143	262	80	158	314	626
	Registers (count)	8	8	21	50	105	218	16	32	64	128
DES-mem	II (cycles)	16	13	9	7	5	5	17	25	41	73
	Area (rows)	69	72	84	143	174	263	141	279	555	1107
	Registers (count)	5	8	19	60	99	174	15	29	57	113
DES-hw	II (cycles)	8	5	5	3	3	2	5	5	5	5
	Area (rows)	27	30	36	56	99	141	57	111	219	435
	Registers (count)	5	8	13	33	73	115	15	29	57	113
IIR	II (cycles)	56	13	29	15	9	5	13	18	33	65
	Area (rows)	106	131	118	138	177	258	253	497	985	1961
	Registers (count)	2	26	14	34	73	154	48	92	180	356

Table 6.2: Raw data – initiation interval (II), area and register count.

The normalized data corresponding to the figures in Table 6.2 is presented in Table 6.3. The *base case* is the original, non-pipelined version of the benchmarks. A detailed analysis of these values follows.

Benchmark		original	pipelined	squash(2)	squash(4)	squash(8)	squash(16)	jam(2)	jam(4)	jam(8)	jam(16)
Skipjack-mem	Speedup	1.00	1.05	1.83	2.44	2.75	3.14	1.91	3.14	4.63	5.03
	Area	1.00	1.16	1.27	1.86	2.92	5.22	2.27	4.47	8.88	17.69
	Registers	1.00	2.17	3.00	7.33	15.33	32.83	4.17	8.17	16.17	32.17
	Speedup / Area	1.00	0.90	1.45	1.32	0.94	0.60	0.84	0.70	0.52	0.28
Skipjack-hw	Speedup	1.00	1.00	1.73	2.71	4.75	6.33	2.00	4.00	8.00	16.00
	Area	1.00	1.00	1.37	2.10	3.49	6.39	1.95	3.85	7.66	15.27
	Registers	1.00	1.00	2.63	6.25	13.13	27.25	2.00	4.00	8.00	16.00
	Speedup / Area	1.00	1.00	1.26	1.29	1.36	0.99	1.03	1.04	1.04	1.05
DES-mem	Speedup	1.00	1.23	1.78	2.29	3.20	3.20	1.88	2.56	3.12	3.51
	Area	1.00	1.04	1.22	2.07	2.52	3.81	2.04	4.04	8.04	16.04
	Registers	1.00	1.60	3.80	12.00	19.80	34.80	3.00	5.80	11.40	22.60
	Speedup / Area	1.00	1.18	1.46	1.10	1.27	0.84	0.92	0.63	0.39	0.22
DES-hw	Speedup	1.00	1.60	1.60	2.67	2.67	4.00	3.20	6.40	12.80	25.60
	Area	1.00	1.11	1.33	2.07	3.67	5.22	2.11	4.11	8.11	16.11
	Registers	1.00	1.60	2.60	6.60	14.60	23.00	3.00	5.80	11.40	22.60
	Speedup / Area	1.00	1.44	1.20	1.29	0.73	0.77	1.52	1.56	1.58	1.59
IIR	Speedup	1.00	4.31	1.93	3.73	6.22	11.20	8.62	12.44	13.58	13.78
	Area	1.00	1.24	1.11	1.30	1.67	2.43	2.39	4.69	9.29	18.50
	Registers	1.00	13.00	7.00	17.00	36.50	77.00	24.00	46.00	90.00	178.00
	Speedup / Area	1.00	3.49	1.73	2.87	3.73	4.60	3.61	2.65	1.46	0.75

Table 6.3: Normalized data – estimated speedup, area, registers and efficiency (speedup/area).

Unroll-and-squash achieves better speedup than regular pipelining, and usually wins over the worse case unroll-and-jam (Figure 6.1). However, for large unroll factors unroll-and-jam outperforms unroll-and-squash by a big margin in most cases. Still, an interesting observation to make is the fact that, for several benchmarks, unroll-and-jam fails to obtain a speedup proportional to the unroll factor for larger factors (Skipjack-mem, DES-mem and IIR). The reason for this is that the increase of the unroll factor proportionally increases the operator count and, subsequently, the number of memory references. Since the amount of memory accesses is limited to two per clock cycle, more memory references increase the II and decrease the relative speedup. Unlike unroll-and-jam, unroll-and-squash does not change the number of memory references – the initial amount of memory references form the lower bound for the minimum II. Therefore, designs with many memory accesses may benefit from unroll-and-squash more than unroll-and-jam at greater unroll factors. Additionally, unroll-and-squash, in general, performs worse on designs with small original II (Skipjack-hw and DES-hw) because there is not much room for improvement.

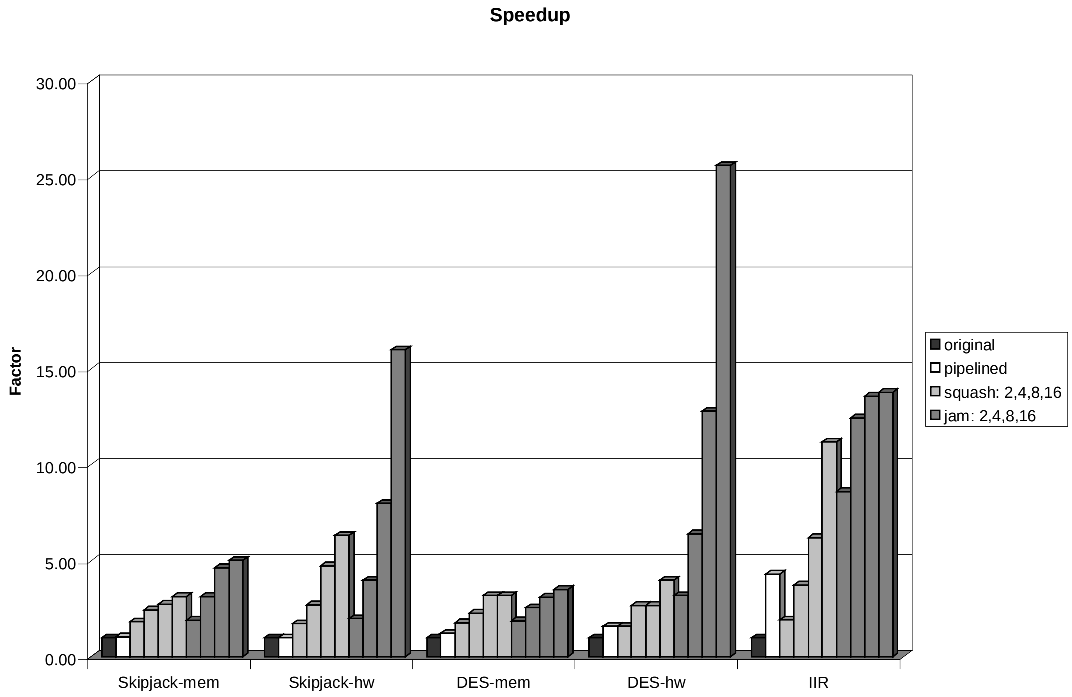

Figure 6.1: Speedup factor.

The speedup from the different transformations comes at the expense of additional area (Figure 6.2). Undoubtedly, since unroll-and-squash adds only registers while unroll-and-jam also increases the number of operators in proportion to the unroll factor, unroll-and-squash results in much less extra area. This can be very clearly seen from the results of the floating point benchmark (IIR) depicted in Figure 6.2.

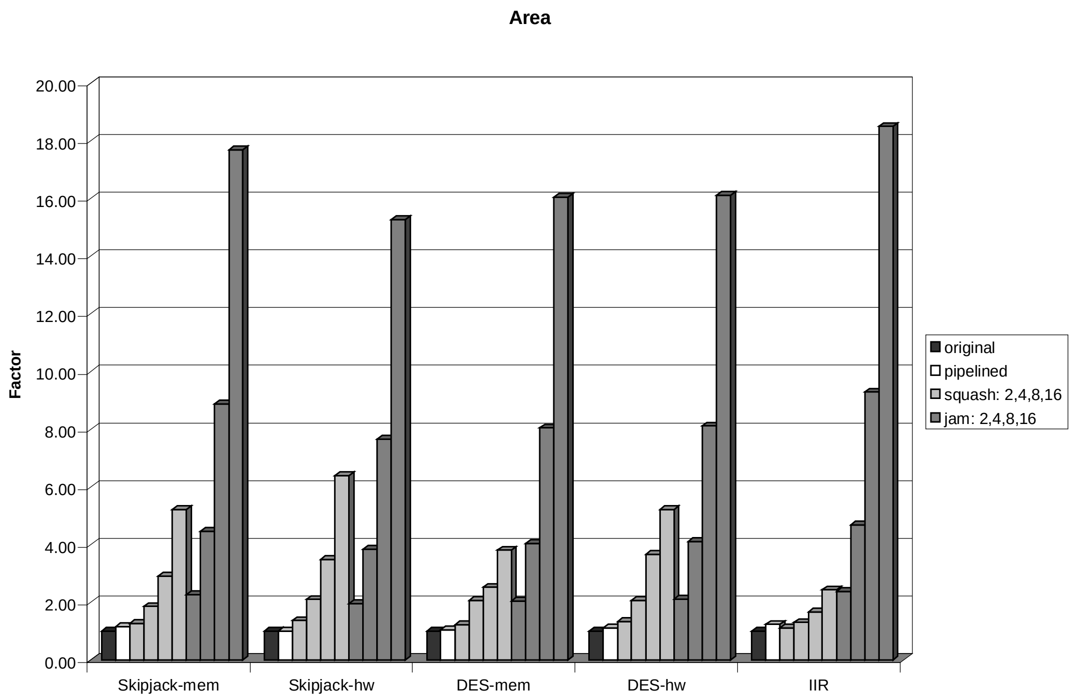

Figure 6.2: Area increase factor.

In order to evaluate which technique is better, we rate the efficiency of the designs by looking at the speedup to area ratio. This value captures the performance of a design per unit area relative to the original version of the loops – a higher speed and a smaller design lead to a larger ratio, while a lower speed and a larger area result in a smaller ratio. Although it is possible to assign different application-specific weights to the performance and the size of a design, these coefficients will only scale the efficiency ratios of the transformed versions, and the relations will remain the same.

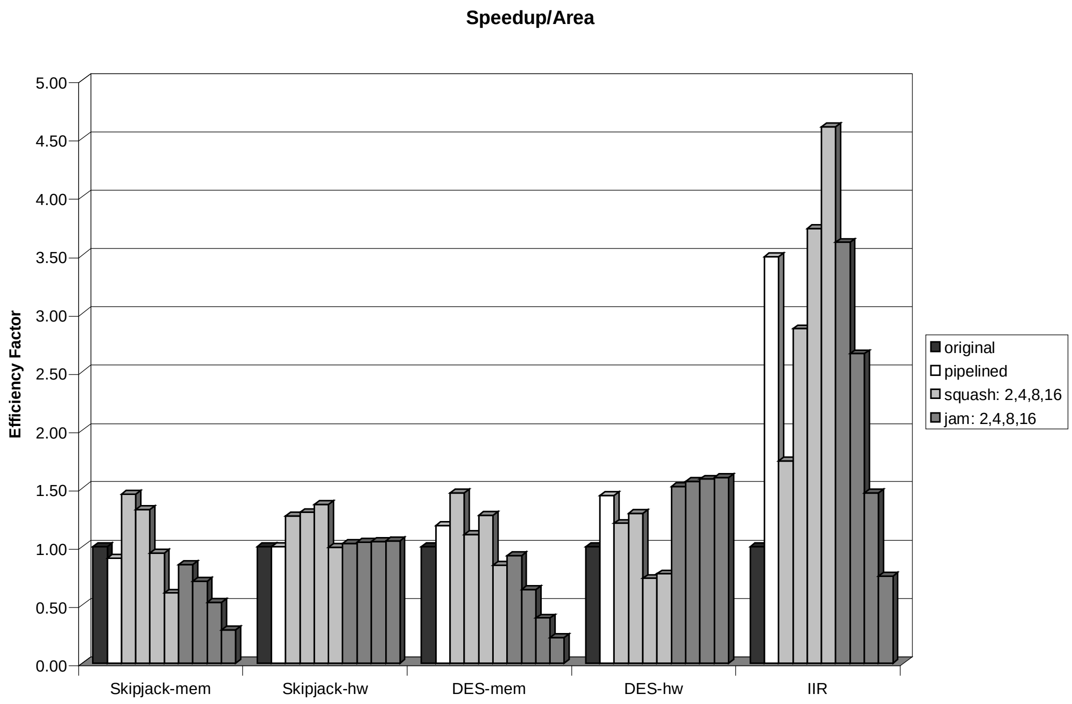

Figure 6.3: Efficiency factor (speedup/area) – higher is better.

By this measure, presented graphically in Figure 6.3, unroll-and-squash wins over unroll-and-jam in most cases, although some interesting trends can be noted in this regard. The ratio decreases with increasing unroll factors when unroll-and-jam is applied to benchmarks with memory references – this is caused by the higher II due to a congested memory bus. However, for designs without memory accesses unroll-and-jam increases the operator count with the unroll factor and does not change the II, so the ratio stays about constant. The efficiency ratio for unroll-and-squash stays about the same or decreases slightly with higher unroll factors in most cases. An obvious exception is the floating point benchmark where higher unroll factors lead to larger efficiency ratios. This can be attributed to the large original II and the small minimum II that unroll-and-squash can achieve – a much higher unroll factor is necessary to reach the point where the memory references limit the II.

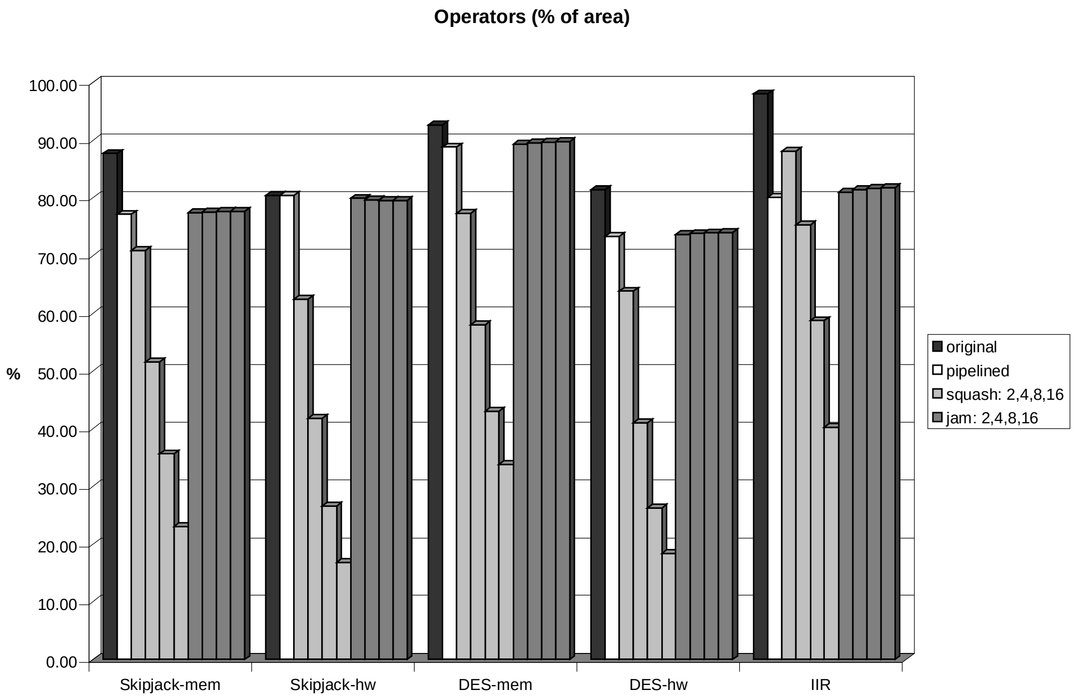

Figure 6.4: Operators as percent of the area.

Finally, it is interesting to observe how the operator count as a proportion of the whole area varies across the different transformations (Figure 6.4). While this value remains about the same for unroll-and-jam applied with different unroll factors, it sharply decreases for unroll-and-squash with higher unroll factors. This is important to note because our prototype implements the registers as regular operators, i. e., each taking a whole row. Considering the fact that they can be much smaller, the presented values for area are fairly conservative and the actual speedup per area ratio will increase significantly for unroll-and-squash in a final hardware implementation. Furthermore, many of the registers in the unroll-and-squashed designs are shift/rotate registers that can be implemented even more efficiently with minimal interconnect.

Chapter 7

Related Work

A large amount of research effort has been concentrated on loop parallelization in compilers for multiprocessors and vector machines [14][29][30]. The techniques, in general, use scalar and array analysis methods to determine coarse-grain parallelism in loops and exploit it by distributing computations across multiple functional elements or processing units. These transformations cannot be effectively applied to hardware synthesis because of the different set of optimization tradeoffs that traditional software compilation faces.

Loop parallelization for uniprocessors involves methods for detection and utilization of instruction-level parallelism inside loops. An extensive survey of the available software pipelining techniques such as modulo scheduling algorithms, perfect pipelining, Petri net model and Vegdahl's technique, and a comparison between the different methods is given in [17]. Since basic-block scheduling is an NP-hard problem [18], most effort on the topic has been focused on a variety of heuristics to reach near-optimal schedules. *Modulo scheduling* algorithms offset the schedule of a single iteration and repeat it in successive iterations for a continuously increasing II until a legal schedule is found. By coupling scheduling with pipelining constraints these techniques easily reach near-optimal schedules and are excellent candidates for software pipelining. While modulo scheduling methods attempt to create a kernel by scheduling a single iteration, *kernel recognition* techniques provide an alternative approach to the software pipelining problem – they schedule multiple iterations and recognize when a kernel has been formed. Window scheduling, for example, makes two copies of the loop body DFG and runs a window down the instructions to determine the best schedule. This technique can be easily combined with loop unrolling to improve the available parallelism. Unroll-and-compact unrolls the loop body and finds a repeating pattern of instructions to determine the pipelined loop body. Finally, *enhanced pipeline scheduling* schemes form the third class of software pipelining algorithms. They combine scheduling with code motion across loop back edges to determine the pipelined loop body along with its prolog and epilog.

The main disadvantage of all these methods when applied to loop nests is that they consider and transform only inner-most loops resulting in poor exploitation of parallelism as well as lower efficiency due to setup costs. Lam's hierarchical reduction scheme pipelines loops that contain control-flow constructs such as nested loops and conditional expressions [19]. To handle nested loops, this method pipelines outward from the innermost loop, reducing each loop as it is scheduled to a single node. Thus, the technique benefits nested loop structures by overlapping execution of the prolog and the epilog of the transformed loop with operations outside the loop. The original Nimble Compiler approach to hardware/software partitioning of loops may pipeline outer loops but considers inner loop entries as exceptional exits from hardware [1]. Overall, the majority of techniques that perform scheduling across basic block boundaries do not handle nested loop structures efficiently [15][20].

Chapter 8

Conclusion

In this thesis we showed that high-level language hardware synthesis needs to employ traditional compilation techniques but most of the standard loop optimizations cannot be directly used. We presented an efficient loop pipelining technique that targets nested loop pairs with iteration-parallel outer loops and strong inter- and intra-iteration data-dependent inner loops. The technique was evaluated using the Nimble Compiler framework on several signal-processing benchmarks. Unroll-and-squash improves the performance at a low additional area cost through efficient resource sharing and proved to be an effective way to exploit parallelism in nested loops mapped to hardware.

Bibliography

[1] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures, *Proc. 37th Design Automation Conference*, pp. 507-512, Los Angeles, CA, 2000.

[2] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing Applications into Silicon. *Proc. IEEE on FCCM*, Napa Valley, April 1999.

[3] T. Callahan, and J. Wawrzynek. Instruction level parallelism for reconfigurable computing, *Proc. 8th International Workshop on Field-Programmable Logic and Applications*, September 1998.

[4] J. R. Hauser, and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor, *Proc. FCCM '97*, 1997.

[5] W. Wolf. Hardware/software co-design of embedded systems, *Proc. IEEE*, July 1994.

[6] B. Dave, G. Lakshminarayana, and N. Jha. COSYN: hardware-software co-synthesis of embedded systems, *Proc. 34th Design Automation Conference*, 1997.

[7] S. Bakshi, and D. Gajski. Partitioning and pipelining for performance-constrained hardware/software systems, *IEEE Transactions on VLSI Systems*, 7(4), December 1999.

[8] R. Dick, and N. Jha. Cords: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems, *Proc. Intl. Conference on Computer-Aided Design*, 1998.

[9] M. Kaul, *et al*. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications, *Proc. 36th Design Automation Conference*, 1999.

[10] M. Gokhale, and A. Marks. Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays, *Proc. FPL*, 1995.

[11] Alpha Data Parallel Systems, *ADM-XRC PCI Mezzanine Card User Guide. Version 1.2*, 1999.

[12] TSI Telsys, *ACE Card Manual*, 1998.

[13] TSI Telsys, *ACE2 Card Manual*, 1998.

[14] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Computer*, December 1996.

[15] Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[16] B. R. Rau, and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *The Journal of Supercomputing*, 7, pp. 9-50, 1993.

[17] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software Pipelining. In *ACM Computing Surveys*, 27(3):367-432, September 1995.

[18] Michael R. Garey, and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., San Francisco, CA, 1979.

[19] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings in SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pp. 318-328, 1988.

[20] Andrew Appel, and Maia Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press, Cambridge, United Kingdom, 1998.

[21] David Ku, and Giovanni De Micheli. *High Level Synthesis of ASICs under Timing and Synchronization Constraints*, Kluwer Academic Publishers, Boston, MA 1992.

[22] SystemC, http://www.systemc.org.

[23] Luciano Lavagno, Ellen Sentovich. ECL: A Specification Environment for System-Level Design, *Proc. DAC '99*, New Orleans, pp. 511-516, June 1999.

[24] Xilinx, http://www.lavalogic.com.

[25] Arvind and X. Shen. Using Term Rewriting Systems to Design and Verify Processors, *IEEE Micro Special Issue on Modeling and Validation of Microprocessors*, May/June 1999.

[26] M. Haldar, A. Nayak, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary and P. Banerjee. A Library-Based Compiler to Execute MATLAB Programs on a Heterogeneous Platform, *ISCA 13th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS-2000)*, August 2000.

[27] Dror E. Maydan. *Accurate Analysis of Array References*, Ph.D. thesis, Stanford University, Computer Systems Laboratory, September 1992.

[28] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[29] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, Prentice-Hall, 1972.

[30] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*, Ph.D. thesis, Stanford University, Computer Systems Laboratory, August 1992.

[31] Michael D. Smith. Extending SUIF for machine-dependent optimizations. In *Proc. of the First SUIF Compiler Workshop*, pp. 14-25, Stanford, CA, January 1996.