

# Efficient Pipelining of Nested Loops: Unroll-and-Squash

Darin Petkov  
Massachusetts Institute of  
Technology  
darin\_petkov@alum.mit.edu

Randolph Harr  
Synopsys, Inc.

Saman Amarasinghe  
Massachusetts Institute of  
Technology  
saman@lcs.mit.edu

## 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. Since streaming data processing in DSP applications 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 paper, we introduce a new method for mapping and pipelining nested loops efficiently into hardware. It achieves fine-grain parallelism even on strong intra- and inter-iteration data-dependent inner loops and, by sharing resources economically, 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 improvement in the area efficiency compared to the best known optimization techniques.*

## 1. Introduction

Growing consumer market needs that require processing of large amount of data with a limited power or dollar budget have led to the development of increasingly complex embedded systems and application-specific IC's. As a result, high-level compilation and sophisticated CAD tools are used to automate and accelerate the intricate design process. These techniques raise the level of abstraction and bring the hardware design closer and closer to the system engineer.

Since loops are the most critical part of many applications (and, specifically, signal-processing algorithms [1]), the new generation of CAD tools needs to borrow from the many traditional compiler transformation methods in order to synthesize hardware from high-level languages [3][4][5]. However, direct application of these techniques fails to produce efficient hardware because the

optimization trade-offs in circuit synthesis from a program and in software compilation to a microprocessor are quite different (for example, code size vs. operator and gate count, register files vs. pipeline registers).

When an inner loop has no cross-iteration data dependencies, many techniques 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 dependencies. Many cryptographic algorithms, such as unchained Skipjack and DES for example, have a nested loop structure where an iteration-parallel outer loop traverses the data stream while the inner loop transforms each data block. For instance, Skipjack (Figure 1) 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 pipelined efficiently. The outer loop, however, has no strong inter-iteration data-dependencies, which allows parallel execution of the separate iterations.

This paper introduces a new loop transformation that maps nested loops following this pattern into hardware efficiently. The technique, which we call *unroll-and-squash*, exploits the outer loop parallelism and concentrates more computation in the inner loop. It improves the performance with little area increase by allocating the hardware resources without expensive multiplexing or complex routing required by the traditional resource sharing methods. The transformation can be applied to any outer iteration-parallel loop to synthesize the inner sequential loops in hardware. It was prototyped using the Nimble Compiler environment [1][2] and evaluated on several signal-processing benchmarks. Unroll-and-squash reaches performance comparable to traditional loop transformations with 2 to 10 times less area.

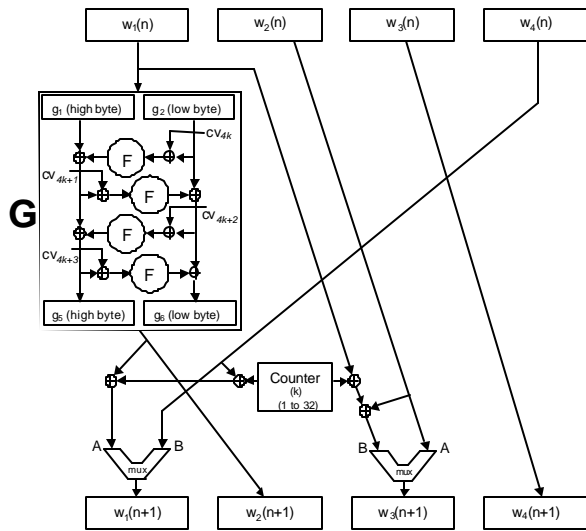


Figure 1. Skipjack cryptographic algorithm

The application of the technique is demonstrated using the simple loop nest in Figure 2. The outer loop walks through the input data and writes out the result, while the inner loop runs the data through several rounds of 2 operators,  $f$  and  $g$ , each completing in 1 clock cycle. The data-dependence cycle carried by the inner loop makes pipelining impossible, i. e., inner loop iterations can't be executed in parallel. 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  $2M\#N$ .

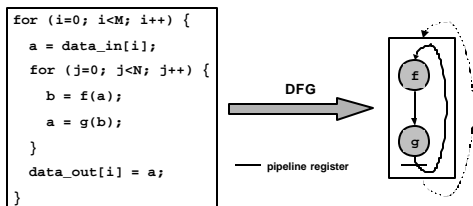


Figure 2 A simple loop nest

Traditional loop optimizations such loop unrolling, flattening and permutation [13] fail to exploit the parallelism efficiently and improve the performance for this loop nest. One successful approach is the application of *unroll-and-jam* (Figure 3), which unrolls the outer loop but fuses the resulting sequential inner loops to maintain a single inner loop [12]. Unroll-and-jam with a factor of 2 (assuming that  $M$  is even) increases the inner loop operators to 4 (twice the original number). The II is still 2 but the total time is half the original because the outer loop iteration count is halved –  $2\#(M/2)\#N=M\#N$ . Thus, unroll-and-jam doubles the performance of the application at the expense of a doubled operator count.

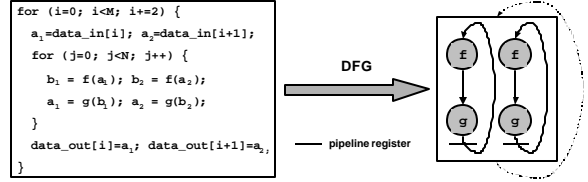


Figure 3. A simple loop nest: unroll-and-jam by 2

A more efficient way to improve the performance in this example is to apply the unroll-and-squash technique introduced in this paper (Figure 4). This transformation, similarly to unroll-and-jam, unrolls the outer loop but maintains a single inner loop. However, the data sets of the different outer loop iterations run through the inner loop operators in a round-robin manner allowing their parallel execution. Moreover, the original operator count remains unchanged. Application of unroll-and-squash to the sample loop nest by a factor of 2 is similar to unroll-and-jam with respect to the transformation of the outer loop – the 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. After adding variable shift/rotate statements and pulling appropriate prolog and epilog out of the inner loop, the transformation can be expressed correctly in software, although this may 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  $1\#(M/2)\#(2\#N)=M\#N$  (ignoring the prolog and the epilog). Thus, unroll-and-squash may almost double the performance without paying the additional cost of extra operators.

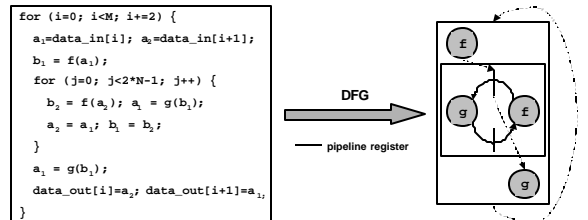


Figure 4. A simple loop nest: unroll-and-squash by 2

It is also possible to combine both transformation techniques. 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 improve further the performance and achieve better operator utilization.

## 2. Method

Unroll-and-squash 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. This section 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*).

### 2.1. Requirements

To apply unroll-and-squash to a set of 2 nested loops with 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 outer loop iterations. The latter condition also implies that the control-flow always passes through the inner loop.

### 2.2. Compiler analysis and optimization techniques

A number of traditional compiler analysis and optimization methods can be used to determine whether a loop nest follows the requirements, to convert it to one that conforms with them, or to increase the efficiency of unroll-and-squash. Unfortunately, few transformations enlarge the set of loops that this technique applies to.

One way to eliminate conditional statements in the inner loop making 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 and  $M \bmod DS$  iterations of the outer loop may be executed independently from the remaining  $M - (M \bmod DS)$ .

The condition that the outer loop iterations are parallel is much more difficult to determine or overcome. Moreover, if the outer loop data dependence is an innate part of the implemented algorithm, it is usually impossible to apply unroll-and-squash. One approach to eliminate some of the scalar data dependencies in the outer loops is induction variable identification – convert all induction variables in the outer loop to expressions of the loop index. Another method is modulo variable expansion, which makes multiple copies of a variable each corresponding to a different iteration and combines them at the end. For loops with array references, dependence analysis [14] 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 methods (such as converting pointer to array accesses) may be utilized to determine whether code with pointer-based memory accesses can be parallelized.

### 2.3. Transformation

Applying unroll-and-squash by a factor  $DS$  to a loop nest requires an efficient schedule of the functional units in the inner loop to separate pipeline stages and a corresponding transformation of the software portion of the loop. Although a hardware-only implementation of the inner loop is possible (without a prolog and an epilog in software), the outer loop still needs to be unrolled and have a proper variable assignment. The basic steps necessary to apply unroll-and-squash are listed below:

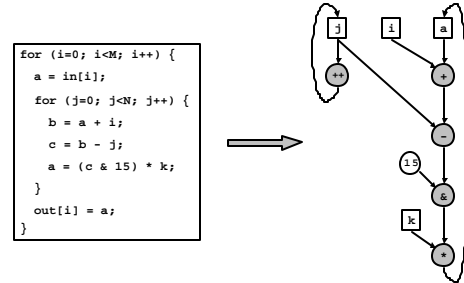


Figure 5. Unroll-and-squash – building the DFG

- Build the DFG of the inner loop (Figure 5). Live variables are stored in registers at the top of the graph.
- Transform live variables used in the inner but defined in the outer loop (registers with no incoming edges) into cycles (edges from the register back to itself).
- “Stretch” the cycles in the graph so that the backedges start from the bottom and end in the registers at the top.
- Ignoring the backedges, pipeline the resulting DFG (Figure 6) producing exactly  $DS$  pipeline stages. Empty stages may be added or pipeline registers may be removed to adjust the stage count to  $DS$ .
- Expand each variable in the nest to  $DS$  copies. Some of the resulting variables may not be necessary later.
- Unroll the outer loop basic blocks (basic blocks that dominate and post-dominate the inner loop).

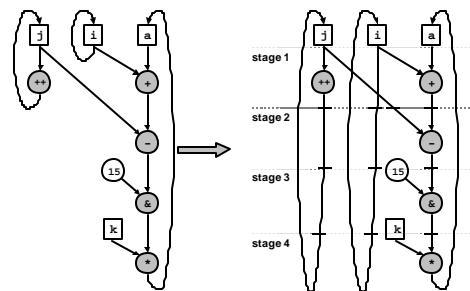


Figure 6. Stretching cycles and pipelining

- Generate prolog and epilog code to fill and flush the pipeline (unless the inner loop is implemented purely in hardware).
- Replace each variable in the inner loop with the copy (from variable expansion) corresponding to the

pipeline stage that the operator is assigned to. Note that some new (delay) variables may be needed to handle expressions split across pipeline registers.

- Add variable shift/rotate to the inner loop. Note that reverse shift/rotate may be required in the epilog or, alternatively, a rotated variable assignment may be used.

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

## 2.4. 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 –  $M \times N$ . The total number of executed operators (except the copy statements that may be free in hardware) stays the same.

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 inner loop pipeline stages. 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 hardware-only implementation bounds the stage count to the critical path delay 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 are processed in parallel, the better the performance. The unroll factor  $DS$  should be calculated in accordance to the outer loop iteration count (loop peeling may be required) and the data dependence analysis discussed in the previous section (larger  $DS$  may eliminate the parallelism).

Another important factor for determining the unroll amount is the extra area and, consequently, extra power that comes with large values of  $DS$ . Unroll-and-squash adds only pipeline registers and data feeds between them and, because of the cycle stretching, most of them can be 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 multiplexers 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 trade-off between speed and area is further illustrated in the benchmark report (Section 4).

## 3. Implementation

The system presented in this paper was developed and evaluated within the Nimble Compiler environment [1][2]. The Nimble Compiler extracts hardware kernels (inner loops that take most of the execution time) from C applications and accelerates them on a reconfigurable coprocessor. It is built upon the SUIF compiler framework [6]. The target architecture (Agile hardware) 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 code by flexible exploitation of ILP.

## 4. Experimental results

We compared the performance of unroll-and-squash on the main computational kernels of several 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.

It is important to note that the prototype implements the registers as regular operators. Considering the fact that registers can be much smaller, the presented values for area are conservative and the unroll-and-squash speedup per area ratio will increase significantly in a final hardware implementation. Furthermore, many of the registers in the transformed designs are shift/rotate registers that can be implemented even more efficiently with minimal interconnect.

### 4.1. Benchmarks

The benchmark suit, described in Table 1, consists of two cryptographic algorithms (unchained Skipjack and DES) and a filter (IIR). 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 hardware – they use local ROM for memory lookups and domain generators for some bit-level operations. IIR is a filter implemented on the target platform by modeling pipelined floating-point arithmetic operations.

We compare 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. 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 have been specially optimized for hardware and have no memory references at all. Some of the normalized data is presented below along with detailed analysis.

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

Table 1. Table 1. Benchmark description.

## 4.2. Results and analysis

While unroll-and-squash achieves better speedup than regular pipelining and usually wins over the worst case unroll-and-jam (Figure 7), with large unroll factors unroll-and-jam outperforms unroll-and-squash by a big margin in most cases. On several benchmarks, however, unroll-and-jam fails to obtain a speedup proportional to the unroll amount for larger factors. The transformation increases proportionally the number of memory references and the II becomes bound by the two accesses per cycle hardware limit. Unroll-and-squash, on the other hand, keeps the number of memory references constant (the initial amount of accesses forms a lower bound for the minimum II) and, therefore, designs with many memory references may additionally benefit from this transformation.

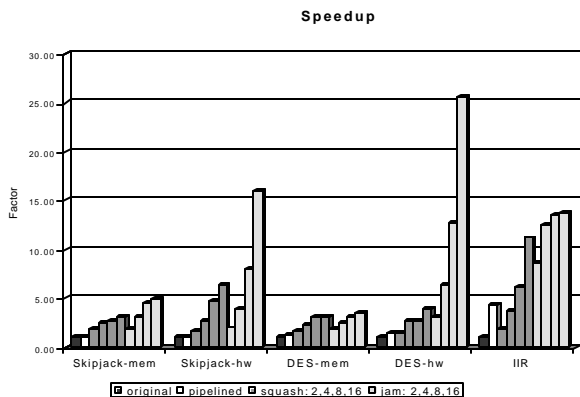


Figure 7. Speedup

The speedup from the different transformations comes at the expense of additional area (Figure 8). Undoubtedly, since unroll-and-squash inserts 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.

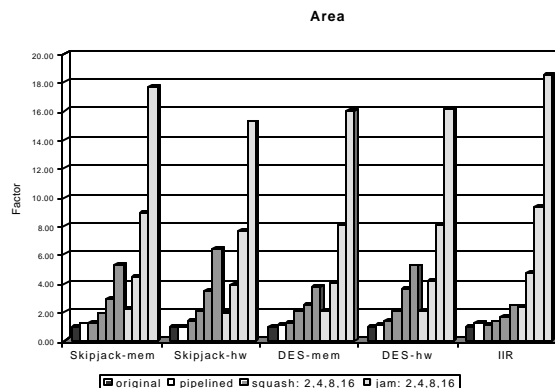


Figure 8. Area

The speedup-to-area ratio (Figure 9) captures the performance of the design per unit area – higher speed and smaller design leads to larger value. By this measure, unroll-and-squash wins over unroll-and-jam in most cases. 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 references unroll-and-jam increases the operator count with the unroll factor and does not change the II, so the ratio stays about constant. The ratio for unroll-and-squash stays about the same or decreases slightly with higher unroll factors with the exception of IIR. The ratio increase in that case 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 to the point where the memory limits the II.

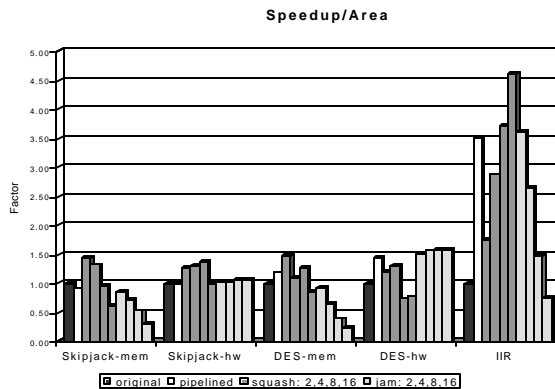


Figure 9. Efficiency (speedup/area) – higher is better

## 5. Related work

An extensive survey of the available software pipelining techniques such as modulo scheduling algorithms, perfect pipelining, Petri net model, and

Vegdahl's technique, as well as a comparison between the different methods is given in [8]. Since basic-block scheduling is an NP-hard problem [9], most work on the topic has been concentrated on a variety of heuristics to reach near-optimal schedules. The main disadvantage of all these methods when applied to loop nests is that they consider and transform only innermost loops resulting in poor exploitation of parallelism and lower efficiency due to setup costs. Lam's hierarchical reduction scheme aims to overlap execution of the prolog and the epilog of the transformed loop with operations outside the loop [10]. 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 few techniques for scheduling across basic block boundaries handle nested loop structures efficiently [7][11]. The general theory of hardware pipelining and optimal scheduling of recurrences can be found in [15].

## 6. Conclusion

This paper presented a loop pipelining technique that targets nested loop pairs with an iteration-parallel outer loop and a strong inter- and intra-iteration data-dependent inner loop. The method 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 resource sharing and proves to be an effective way to exploit parallelism in nested loops mapped to hardware. It can be used as a valuable and simpler alternative to complex loop pipelining schemes that enables efficient hardware synthesis from high-level languages.

## 7. Acknowledgement

This work was completed at the Advanced Technology Group at Synopsys, Inc. with additional support from DARPA's ACS program under AFRL contract #F33615-98-2-1317. The authors would like to thank our collaborators at Lockheed Martin ATL and UC Berkeley.

## References

[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] D. Petkov. *Efficient Pipelining of Nested Loops: Unroll-and-Squash*, M.Eng. thesis, Massachusetts Institute of Technology, January 2001.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] V. Allan, R. Jones, R. Lee, and S. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367-432, September 1995.

[9] M. Garey, and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., San Francisco, CA, 1979.

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

[11] A. Appel, and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, Cambridge, United Kingdom, 1998.

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

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

[14] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, MA, 1989.

[15] P. Kogge. *The Architecture of Pipelined Computers*, McGraw Hill, NY, 1981.