

Sea Cucumber: A Synthesizing Compiler for FPGAs

Justin L. Tripp, Preston A. Jackson, and Brad L. Hutchings

Brigham Young University, Provo, Utah

Abstract. Sea Cucumber (SC) is a synthesizing compiler for FPGAs that accepts Java class files as input (generated from Java source files) and that generates circuits that exploit the coarse- and fine-grained parallelism available in the input class files. Programmers determine the level of coarse-grained parallelism available by organizing their circuit as a set of inter-communicating, concurrent threads (using standard Java threads) that are implemented by SC as concurrent hardware. SC automatically extracts fine-grained parallelism from the body of each thread by processing the byte codes contained in the input class files and employs conventional compiler optimizations such as data-flow and control-flow graph analysis, dead-code elimination, constant folding, operation simplification, predicated static single assignment, if-conversion, hyper-block formation, etc. The resulting EDIF files can be processed using Xilinx place and route software to produce bitstreams that can be downloaded into FPGAs for execution.

1 Introduction

The principal challenge facing developers of synthesizing compilers for FPGAs that are based on sequential programming languages is the discovery and extraction of sufficient parallelism. FPGA systems typically operate at a clock rate less than 1/10 that of a processor fabricated in the same process. In order to overcome this clock rate penalty, an FPGA circuit must exploit more than *an order of magnitude* more available parallelism than a conventional software compiler. Extracting parallelism from sequential descriptions is already a very difficult problem. Requiring that the compiler automatically extract even more parallelism will only make the problem that much more difficult.

SC takes a pragmatic approach to the problem of extracting parallelism. SC requires that programmers expose coarse-grained parallelism by organizing their application as standard Java threads that intercommunicate using provided library functions. SC then analyzes the body of each thread and uses a variety of compiler and circuit optimization techniques to extract fine-grained parallelism. Experience has shown that programmers can do a reasonable job of manually organizing software as relatively coarse units that operate in parallel[1]. In addition, there is a great body of work that deals with automated discovery and exploitation of fine-grained parallelism in the compiler literature[2]. It is believed that by combining these two approaches (programmer-exposure of coarse-grained

parallelism and automated discovery of fine-grained parallelism), SC can produce FPGA circuitry that exploits much more available parallelism.

SC is a work in progress and this paper gives a brief overview of its current status, including an overview of the programming model, presentation of analysis techniques for extracting fine-grained parallelism, and suggestions for future work.

2 Related Work

Compiling from Java byte codes is not new. Past efforts, including GALADRIEL and NENYA[3] and a previous BYU effort[4], compile using byte-codes as their input. Collectively, these efforts indicate the advantages of using byte codes. The advantages include, among others, the lack of writing and maintaining a parser, access to standard compiler and debugging tools, and access to a wide range of available byte-code analysis and modification tools.

SC significantly differs from these projects in various ways. First, unlike GALADRIEL, SC directly generates EDIF files as output. GALADRIEL generates VHDL as output and thus requires the use of an additional synthesis tool to generate EDIF files. It should be noted that because SC performs only limited circuit optimizations, the ultimate quality of circuits could be lower with SC than with a mature synthesis tool. We feel that this discrepancy is assuaged by the ease of working with SC as a single integrated tool. Working with two separate synthesis tools forces the user to be conversant in both to debug problems. Second, unlike the earlier work at BYU, SC programs can be verified by compiling the Java files with the standard Java compiler and executing the class-files in the standard Java Virtual Machine. This feature eliminates the need for circuit simulation to verify behavior as used in the earlier BYU effort.

Snider et al. [5] were the first to articulate the general idea of using a general-purpose language to describe programmer-exposed coarse parallelism and automatic discovery of fine-grained parallelism to an FPGA audience. SC has adopted this coarse/fine approach along with the use of predicated static single assignment (PSSA)[6] and other compiler optimizations described by Snider et al. However, their coarse-level parallel programming model uses a custom approach based on C++ and inheritance whereas SC adopts the standard Java thread model and augments it with a communication model based on Communicating Sequential Processes (CSP)[7]. In addition, SC uses hyperblocks and if-conversion optimizations that are not discussed by Snider et al. and provides ways to control the bit-width of variables in Java.

The Garp project[8] applies some of the techniques and data structures found in VLIW compiler research to FPGA synthesis. In particular, Garp isolates a critical path in a code segment and synthesizes it to a reconfigurable co-processor. SC uses many of the same techniques and data structures but differs from Garp in the extent to which they are used. SC extends the critical-path synthesis performed in Garp to entire system synthesis with multiple paths and multiple parallel threads.

3 Programming Model

The general programming model used by SC is based on Java. Users write circuit descriptions in Java, exposing coarse-level parallelism as concurrent threads that communicate via protocols that are based on CSP. The hardware and software implementations of the communicating functions are provided by SC. Bit-precision of variables is specified by the programmer using a `BitWidth` package that is also provided as a part of SC. SC also performs bit-precision analysis following the specification to eliminate unnecessary precision of intermediate signals. Finally, programmers verify the behavior of their application by compiling their Java source and executing and debugging it using standard Java Virtual Machines (JVM) and debuggers—no simulation is necessary. Once satisfied with the behavior, the programmer invokes the SC compiler on the relevant class file to generate an EDIF netlist and further invokes Xilinx place and route software to create a bitstream from the synthesized EDIF netlist. SC supports standard Java class-files; however, there are some restrictions. Use of the `new` operator, object references, static native Java calls (e.g. `System.out.println()`), string manipulation, and exception handling are restricted in the current version of SC.

Java was chosen to be the source programming language for SC because it is popular, portable, widely available, and a generally more productive programming environment when compared to C or C++. SC designs can be compiled, debugged, and simulated with standard tools that are freely available on the internet. Also, unlike C/C++, threads are a part of the language specification making it easier to exploit a programmer base that already understands the Java threading model.

As discussed above, Java threads are used by the programmer to express coarse-level parallelism. Typically, in software, threads give the illusion of concurrency for programming expediency, or limited concurrency if multiple processors are available. However, when compiled by SC, each thread is implemented using dedicated hardware thereby guaranteeing full concurrency. In order to facilitate thread hardware synthesis, it was necessary to create an inter-thread communication scheme that eliminates shared memory access, by limiting all communication to a specified channel. This scheme is based on CSP.

CSP was chosen for inter-thread communication because it provides a programmable, well-behaved model for communication. Handel-C[9], and SystemC[10] are other tools that provide CSP as a communication model. Hoare's CSP[7] provides a theoretical basis for consistent and provable behavior. The provided CSP channels simplify communication and free the programmer to concentrate on the design and behavior of threads.

As described in [11], channels connect two threads and manage the communication of data between them. A channel is unbuffered, one-way, and self-synchronizing. Communication in both directions between two threads requires two channels. Both sending and receiving threads must be ready before code execution will continue in either thread. Since the threads synchronize around the channel communication, it is the responsibility of the programmer to avoid deadlock.

BitWidth is a library of variable precision functions for exactly specifying and simulating the width in bits of integer and floating-point values and operations in Java. BitWidth provides function calls which indicate the sizes of variables in the Java source code. The functions calls become markers in the classfile after the Java source is compiled. After compilation, BitWidth processes the classfile and produces a new classfile in which the operations on the variables have been replaced with bit-corrected versions. The BitWidth produced version can be executed to test the behavior of the code with the correct variable bit-width. The bit-width markers are used during synthesis to determine the size of registers and wires.¹

4 Sea Cucumber Compilation and Synthesis Process

SC synthesizes from Java class files into EDIF. Figure 1 outlines the different steps involved in SC. The basic steps are as follows: pre-processing, byte-code analysis, extracting fine-grain parallelism, compiler optimizations, and net-list generation.

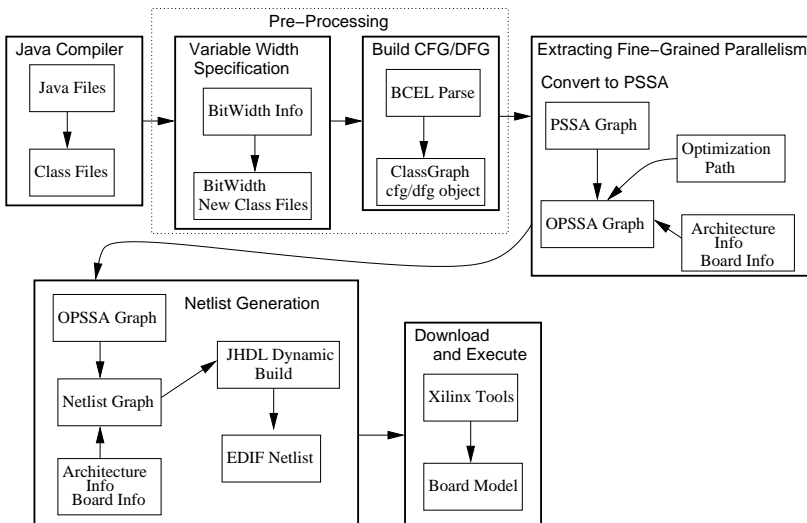


Fig. 1. Overview of Sea Cucumber

4.1 Preprocessing

Preprocessing involves two steps: bit-width analysis to obtain the bitwidth behavior as described in section 3, and class-file stitching which determines the relationship of different class files.

¹ Due to a re-write of the interface layer to the byte code parser, SeaCucumber does not currently use BitWidth information. BitWidth can be used for bit-correct simulation, but a static data structure is used to communicate the sizes of variables.

In this stage of SC, bit-width analysis locates the bit-width markers in the byte code and records them for use in netlisting and optimization. The bit-width markers are found by parsing the byte-code, locating the function call signatures, and parsing the inputs to the function calls. If the byte code has not been updated with the BitWidth package, then the natural Java variable size is used. (e.g. a `byte` is eight bits, an `int` is thirty-two bits, etc.)

The class-file sticher reads the input class file and parses the byte code of the `main()` method in order to determine the structure of the SC design. The constructors of the different channels and threads are located and the inputs to the constructors are determined. This information is passed back to SC, so that all of the threads can have their byte-code parsed and the channels can be connected properly.

<pre> if (a < b) { c = d + e; } else { if (a < 0) { c = d - e; } else { c = c * 2; } } </pre> <p style="text-align: center;">(a) Code</p>	<pre> if (a_1 < b_1) { c_2 = d_1 + e_1; } else { if (a_1 < 0) { c_3 = d_1 - e_1; } else { c_4 = c_1 * 2; } } c_6 = φ(c_2, c_3, c_4) </pre> <p style="text-align: center;">(b) SSA</p>
---	---

Fig. 2. Example Code and SSA Conversion

4.2 Byte-Code Analysis

SC disassembles the Java class files in order to perform control-flow and data-flow analysis using the same static analysis and emulation of Java byte code as is done in [4]. The result is a control-flow graph (CFG) which contains vertices called basic blocks. A basic block is a sequence of straight-line code that has a single entry point at the beginning and a single exit point at the end[2]. Each basic block contains a data-flow graph, which is a directed-acyclic graph representing the operations and data dependencies of that basic block. The CFG and DFGs are further modified to expose greater parallelism.

As part of the DFG generation, all references to variables are uniquely numbered using Static Single Assignment (SSA). In SSA, each variable can only be assigned once. Each new assignment to a variable creates a unique variable name. As can be seen in figure 2(b), the first assignment to variable `c` is called `c_2`, the second assignment `c_3`, and so on. Operations in this format can be scheduled as soon as their real data dependencies have been met.

Once the DFGs and CFGs are formed, the DFGs are traversed in order to form ordered lists of operations. The traversal is accomplished by recursively visiting every vertex in the DFG, determining its type and creating a corresponding operation in the list. DFGs for channel communication receive special

handling, since the hardware has already been laid out by hand. An example of an operation graph is in figure 3(b).

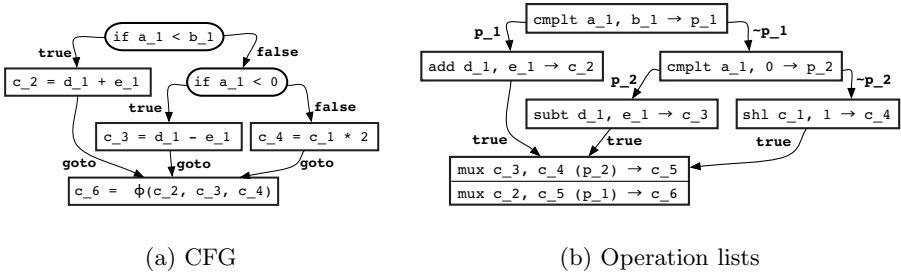


Fig. 3. Creation of the DFG and CFG

4.3 Extracting Fine-Grained Parallelism

SC uses the hyperblock to maximize fine-grained parallelism. A hyperblock, described in [12], is an extended basic-block of instructions which has one input path but may have any number of output paths. To form hyperblocks, SC uses instruction predicates[6] and if-conversion[13]. An instruction predicate is a field which indicates whether or not an instruction should be executed. If-conversion is the process of forming these predicates from if-statements and edges in the CFG. It allows instructions which fall into parallel if- and else-blocks to be executed in parallel. After execution, the results of both paths are multiplexed together and the correct result is selected using the instruction predicate.

The goal of hyperblock formation is to combine as many instructions together as possible. Having a large number of instructions in each hyperblock enables greater fine-grained parallelism and optimization. In practice, however, the amount of exploitable parallelism is ultimately limited by the amount of available hardware. For example, in VLIW processors, the number of functional units determines how many operations can proceed in parallel. The size of the hyperblock must be limited using heuristic algorithms based on execution profiles. In contrast, SC assumes that infinite hardware resources are available and forms hyperblocks of maximum size, thus ensuring a greater degree of parallelism.

In order to form hyperblocks, all of the operations² in the graph must be predicated. Before if-conversion, each vertex in the CFG contains a list of unpredicated operations extracted from the DFG. These vertices are connected by edges which represent branches in the control-flow. Each edge is labeled with the boolean expression which indicates when the branch is taken. For example, in figure 3(b) the labels on the edges are derived from the results of the compare operations, p_1 indicates that the branch taken when the first instruction evaluates to **true**. Likewise, $\sim p_1$ indicates the branch taken when the first operation is **false**. SC forms predicates for the operations using these edge expressions

² In SC, we use the term *operation* instead of *instruction* because each operation is explicitly built in hardware.

during if-conversion. When if-conversion is complete, the information stored on each forward edge in the CFG is also stored in each operation's predicate.

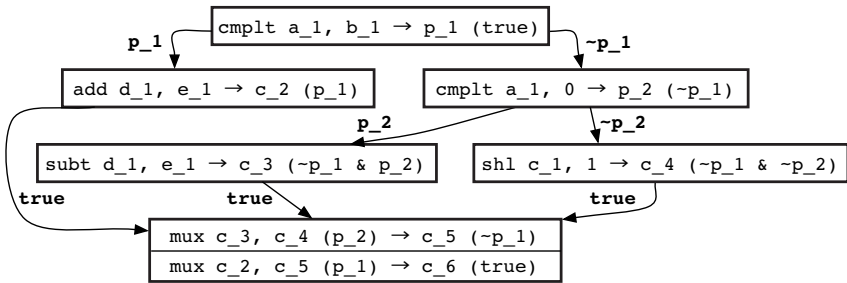


Fig. 4. Predicated Operations

If-conversion was originally written to exploit parallelism for vector processors. In SC, it is used to form operation predicates. During if-conversion, each edge in the graph is tagged as a forward or backward edge. A forward edge represents a branch to a later section of code and a backward edge represents a branch to an earlier section of code. The edge labels are used to form predicates in all of the operations which follow it in the graph. The edge label is added to each operation predicate across multiple basic-blocks until a backward edge is found. In figure 4 the `shl` instruction has a predicate of $(\sim p_1 \ \& \ \sim p_2)$. This predicate means that this instruction follows the $\sim p_1$ and $\sim p_2$ branches in the graph. Terminating at backward edges prevents the endless application of predicates around a loop in the CFG. An example of added predicates can be seen in figure 4. Each operation contains the value indicating when its result is valid.

Once the edge expressions of each forward edge are retained in the operation predicates, SC forms hyperblocks by removing the forward edges. There are two ways to merge basic-blocks into hyperblocks. First, if a basic-block has two children, representing two alternative execution paths, they can be merged together into a single hyper-block. This type of merging is called a parallel-merge. Second, when two blocks fall in a single execution path, they can be merged using serial-merging. When the merging is complete the forward edges have been removed and the basic-blocks are replaced with hyperblocks. The difference can be seen between figure 4 and figure 5. In figure 5 all of the forward branches have been removed and the instructions all reside in a single hyperblock.

After merging basic blocks, each new merged block will contain multiple operator lists that must also be merged. This merging step is complicated by input and output considerations. Consider outputs – when merging two lists in parallel, the outputs of both lists must be unique. If not, then a mux operator must be added to select the correct value from the two outputs. When merging two lists serially, similar problems arise. For example, if the second list reads a variable from the first list, the first output becomes a local variable to the new merged list. Also, inputs common to both lists must read from the same variable and outputs must be unique.

cycle	operation	predicate
0	cmplt a_1, b_1 → p_1	(true)
0	add d_1, e_1 → c_2	(p_1)
0	cmplt a_1, 0 → p_2	(¬p_1)
0	subt d_1, e_1 → c_3	(¬p_1 AND p_2)
0	shl c_1, 1 → c_4	(¬p_1 AND ¬p_2)
1	mux c_3, c_4 (p_2) → c_5	(¬p_1)
2	mux c_2, c_5 (p_1) → c_6	(true)

Fig. 5. Hyperblock

4.4 Compiler Optimizations

Sea Cucumber uses compiler optimizations to simplify the code before it is synthesized. As in [5] optimizations are performed at different levels of abstraction. In the compiler, optimizations occur at the operation level, the hyper-block level and the at the graph level.

Optimizations at the operation level examine only a single operation. Operation simplification and constant folding can be performed at this level. Typically the Java compiler will take care of constant math operations, but these operations can be created as side-effects of the SC programming style or other optimizations. Operation simplification converts an operation to one that is simpler to implement for the given platform. Examples of such optimizations are additive and multiplicative identities, multiplying or dividing by 2^i , and multiply or shifting by zero.

Hyper-block optimizations consider an entire hyper-block and examples of these optimizations are the dead-code removal, and scheduling. Dead-code occurs for several different reasons: side-effects of optimization, hyper-block formation, channel communication and array formation. Through the optimization process, dead-code is located and the predicate of those operations will be set to **false**. These invalid operations are removed from the hyperblock, but this must be done iteratively in order to remove all operations that contributed to the removed operation. The removal of this dead-code can result in an empty hyperblock. The empty hyperblock is removed from the graph and the control flow is updated appropriately.

Scheduling is one of the last optimizations to be performed. Scheduling is required for circuit generation and state-machine creation. Currently, SC schedules operations by dependence. As soon as the inputs to an operation are made available, the operation will be scheduled. The scheduler calculates the cycle in which a given operation must be executed (see figure 5). With this information a state-machine for the given thread can be created. The state machine controls flow of execution in the hyperblock and is used to determine which and when outputs of operations will be written to registers.

4.5 Net-List Generation and Bit-Stream Synthesis

Net-list generation is the process by which the SC intermediate form, a CFG with PSSA operations, is converted into a generic circuit object. PSSA operations in

a hyper-block lend themselves to direct synthesis into the netlist. State machines are generated by examining the operations and creating the circuitry necessary. CSP channels are implemented with custom-built circuits. SC produces an EDIF netlist which can be used as the input to bitstream synthesis tools.

State machines are used in SC to determine the flow of execution between hyperblocks, and control when variable registers are updated. In a thread, only one hyperblock is active. The active hyperblock is responsible for calculating which succeeding hyperblock should be activated next. The state machine generator examines the exit edges of a hyperblock and correlates the conditions of exit from the predicates found in that hyperblock. For example, if all future operations are predicated A and the exit edge is predicated $\sim A$, then the state machine generator will know that in the cycle where A is calculated, the future hyperblock at the end of the $\sim A$ edge can be sent a start signal. If the exit edge points to the same hyperblock from which it came (a loop), the hyperblock will return to its initial state.

After the state machine has been generated, SC generates data-path elements using the hyperblock's operation list. Each operation contains an operator, a list of operands, and a predicate. All operators are directly synthesized using module generators or logical operations. The operation's operands represent wires in the netlist. Because these operand names are generated in PSSA form, each wire has a unique name. This transformation can be seen in figure 6.

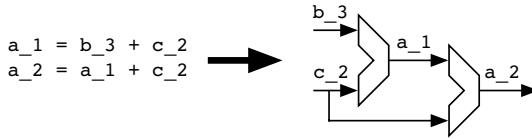


Fig. 6. Synthesis of hyperblock operations

The operation predicates are synthesized using boolean logic gates. The output of these gates is used in two cases. First, when a result of a computation must be saved for use in other hyperblocks, the output acts as the enable line to the storage register. Second, the output is used by the state machine to control the next-state transitions and early-exits of the hyperblock.

The CSP channels synthesize to hardware in a straightforward fashion. On each side of the channel a write and read operation blocks until its complementary operation is ready. This behavior is implemented easily using simple waiting state machines on both sides of the channel. The channel circuitry is simply an interface to which both threads can connect to transfer data. When both operations are ready and the data has been transferred, the state machines release control of the thread and execution proceeds.

In the SC project, we do not tackle the problem of generating a configuration file for the FPGA, rather we break the tool chain and the user must run the platform specific configuration-file generation tools. SC produces an EDIF netlist which can be processed using manufacturer-provided place and route software to produce bitstreams that can be downloaded into FPGAs for execution. Once

this side step in the tool chain has been completed the design is ready to run in hardware.

5 Closing Remarks

Sea Cucumber brings a novel programming model to FPGA synthesis. Using standard Java threads to expose coarse-grained parallelism and VLIW techniques to increase fine-grained parallelism, SC provides a paradigm to exploit large amounts of parallelism. Different from some other tools, programming and simulating a design happens in the same environment.

SC is a work in progress. It currently produces simulatable JHDL designs and can netlist EDIF. We are working on the seamless support for a Virtex II board so that designs can be downloaded to hardware. Future work will explore the impact of programming and synthesizing other communication models and determining which optimizations bring the greatest levels of parallelism.

References

- [1] M. P. I. Forum, "MPI: A message-passing interface standard," Tech. Rep. UT-CS-94-230, 1994.
- [2] S. S. Munchnick, *Advanced Compiler Design and implementation*. San Francisco, California, USA: Morgan Kaufmann Publishers, Inc., third ed., 1997.
- [3] J. M. P. Cardoso and H. C. Neto, "Macro-based hardware compilation of java bytecodes into a dynamic reconfigurable computing system," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines* (K. L. Pocek and J. M. Arnold, eds.), (Napa, CA), p. n/a, IEEE, 1999.
- [4] M. J. Wirthlin, B. L. Hutchings, and C. Worth, "Synthesizing rtl hardware from java byte codes," in *Field Programmable Logic and Applications: 11th International Conference proceedings / FPL 2001*, pp. 1–10, Springer-Verlag, 2001.
- [5] G. Snider, B. Shackelford, and R. J. Carter, "Attacking the semantic gap between application programming languages and configurable hardware," in *FPGA 2001*, (Monterey, CA), pp. 115–124, ACM, ACM, February 2001.
- [6] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante, "Predicated static single assignment," in *IEEE PACT*, pp. 245–255, 1999.
- [7] C. A. R. Hoare, *Communicating Sequential Processes*. London, UK: Prentice-Hall, 1985.
- [8] T. Callahan and J. Wawrzynek, "Adapting software pipelining for reconfigurable computing," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, (San Jose, CA), ACM, 2000.
- [9] Celoxica, *Handel-C Language Reference Manual*. Celoxica Limited, 2001.
- [10] S. Swan, D. Vermeersch, D. Dumlugöl, P. Hardee, T. Hasegawa, A. Rose, M. Coppolla, M. Janssen, T. Grötter, A. Ghosh, and K. Kranen, *Functional Specification for SystemC 2.0*. Open SystemC Initiative, 2.0-p ed., October 2001.
- [11] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers, "Communicating Java Threads," in *Parallel Programming and Java, Proceedings of WoTUG 20* (A. Bakkers, ed.), vol. 50, (University of Twente, Netherlands), pp. 48–76, IOS Press, Netherlands, 1997.

- [12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, “Effective compiler support for predicated execution using the hyperblock,” in *25th Annual International Symposium on Microarchitecture*, 1992.
- [13] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren, “Conversion of control dependence to data dependence,” in *Symposium on Principles of Programming Languages*, pp. 177–189, 1983.