Automatic Compilation of C for Hybrid Reconfigurable Architectures

by

Timothy John Callahan

B.S. (University of Minnesota) 1990
Diploma in Computer Science (University of Cambridge, England) 1991
M.S. (University of California at Berkeley) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor John Wawrzynek, Chair
Professor Alex Aiken
Professor Dorit Hochbaum

Fall 2002

The dissertation of Timothy John Callahan is approved:

Chair	Date
	Date
	Date

University of California, Berkeley

Fall 2002

Automatic Compilation of C for Hybrid Reconfigurable Architectures

Copyright 2002

by

Timothy John Callahan

Abstract

Automatic Compilation of C for Hybrid Reconfigurable Architectures

by

Timothy John Callahan

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

Microprocessors coupled with dynamically reconfigurable hardware offer the potential of performance approaching that of an application-specific integrated circuit, but with the reprogrammability and mass production cost benefits of a microprocessor. My research group has designed such a processor, called Garp. Throughout program execution Garp's coprocessor is reconfigured as necessary at the entrance of each coprocessor-accelerated loop.

This thesis describes `garpcc`, a fully operational prototype C compiler targeting the Garp architecture, automatically utilizing the reconfigurable coprocessor when beneficial with no need for guidance from the programmer.

`Garpcc` uses the hyperblock framework to combine a candidate loop's commonly executed paths using predicated and speculative execution to expose instruction level parallelism (ILP). The hyperblock also allows the exclusion of rarely taken paths, leading to many benefits: coprocessor-infeasible operations on excluded paths do not interfere with acceleration of the rest of the loop; the remaining kernel is smaller allowing it to fit onto the coprocessor's reconfigurable resources when it would not otherwise; and excluding rare paths can improve the performance of remaining paths by removing long dependence chains and improving optimization opportunities.

The hardware synthesis phase of `garpcc` uses a fully-spatial approach where each operation is directly instantiated in the datapath configuration. The spatial approach allows merging of operations into optimized modules. The module-mapping step is timing-sensitive in that it op-

timizes for the critical path; it also simultaneously performs relative placement of the resulting modules. Finally, the spatial datapath is easily pipelined, further increasing the ILP.

The automatic compilation path combined with a cycle-accurate simulation of a realistic implementation of the Garp chip allows quantitative evaluation of the current Garp/`garpcc` system across a number of large benchmarks. Identification of weaknesses is mainly confined to `garpcc`, since until these are addressed, it is premature to fault Garp or its approach to reconfigurable computing in general.

`Garpcc` is shown to create pipelined datapaths capable of exploiting large amounts of ILP. But in many cases large peak ILP does not translate to significant net speedup because with short-running loops the pipeline does not completely fill, and also the overhead of using the coprocessor is more significant. Many improvements to the compilation path are suggested.

Professor John Wawrzynek
Dissertation Committee Chair

To my parents,

Chris and Eileen,

for their unwavering support and patience.

Contents

List of Figures

List of Tables

Acknowledgments

Over the years of my thesis work I have had the privilege and benefit of interacting with a great many academics at other institutions and companies. In particular, Seth Goldstein, Mihai Budiu, Maya Gokhale, Markus Weinhardt, Wayne Luk, Osvaldo Colavin, Alan Marshall, Steve Trimberger, and Dan Poznanovic among many others gave feedback and encouragement that was especially welcome given that I had no collaborators at Berkeley.

I spent the summer of 1998 working at Synopsys. I thank the other members of the just-created Nimble Compiler group for making my time fun and interesting: Randy Harr, Ervan Darnell, Ingo Schaefer, Jonathan Stockwood, Yanbing Li, Uday Kurkure, and Mahadevan Ganapathi a.k.a. Gana. I owe a great thanks to 'Landlady' Jin Liang for providing me with not just housing but also friendship that summer. She and her visiting parents provided great hospitality, food, and card playing fun.

I owe gratitude as well to many in the SUIF community—those using the free compiler infrastructure originating from Stanford. Chris Wilson in the beginning and now Glenn Holloway have continually amazed me with their patience and knowledge in answering any question, no matter how stupid or complex, whether from me or others on the SUIF mailing list. Chris Colohan has also provided a great service by providing and collecting bug fix patches. The CFG representation and profiling strategy used by `garpcc` is based on code [HY97] from Glenn Holloway and Cliff Young of Mike Smith's HUBE Research Group at Harvard. Their libraries also provide the variable liveness analysis used by `garpcc`. Corinna Lee's group at Toronto provided an example pass showing how to move the results from SUIF's dependence library to useful annotations on each memory access.

Purely social friends also played a large part in surviving and enjoying the San Francisco Bay Area, starting with Minnesota connections Hieu Nguyen, Dawn Tilbury, and David Ofelt, and Churchill friends Leslie Phinney, Margaret Guell, David Schwartz, and Michelle Wang. The Alexis Group, a sequel to the Happy Hour list started by Dawn and with too many rotating members to mention, also played an important role. Other important friends in and out of the department include Ngeci Bowman, Jennie Chen, Jeannie Chiu, Mike Shire, and Tina Wong. The list of friends who've

lived with me on Oxford Street has grown long over the years: Desmond Kirkpatrick, Eric Felt, Chris Lennard, Nathan Bossett, Ravi Gunturi, Matt Brukman, David Gelbart, and David Vocadlo.

My longest running recreation has been playing in the EECS department softball league. Its best feature is that you eventually get to know everyone on every team, which unfortunately precludes any exhaustive listing. But I will thank those who have acted as League Czars or as my team captains (when I wasn't captain), starting with Karl Petty, the original instigator; Chad Yoshikawa; Dennis Sylwester; Noah Treuhaft; Steve Chien; and finally, Dennis Geels.

It was a pleasure being part of the BRASS group headed by John Wawrzynek, later assisted by Andre Dehon doing his post-doc. I especially thank fellow grad students John Hauser and Krste Asanovic for their senior guidance. Officemates providing friendship mixed with varying measures of running political commentary and hiking advice include Su-Lin Wu, Christoforos Kozyrakis, Stelios Perissakis, Joe Gebis, and Sam Williams.

Finally I thank my family for their patience and support.

Chapter 1

Introduction

1.1 Reconfiguration for instruction level parallelism

Continuing improvements in microprocessor performance are due in part to microarchitecture techniques that decrease the effective cycles per operation, or conversely, increase the effective operations per cycle. Execution of independent operations in parallel is a key technique along these lines. Dynamically-scheduled superscalar processors [Tho64, Tom67, Joh91] contain control logic to identify independent operations from a window of upcoming instructions. VLIW (very long instruction word) processors [HT72, Cha81, Fis83] instead rely on the compiler to bundle groups of independent operations into long instructions. With vector processors [Rus78], a single instruction instigates multiple identical operations element-wise on vectors of data.

These approaches have their relative strengths and weaknesses, but they all rely on the von Neumann instruction fetch-and-execute model. With this model, there are practical implementation limits on the number or at least variety of operations executed each cycle. Furthermore, instruction bandwidth as well as code density considerations necessitate a limited instruction set that can be encoded into a small number of bits. While instruction sets are complete in that any computation can be performed, it may take a long sequence of instructions to accomplish a given task.

Reconfigurable computing describes a broad class of techniques for bypassing the limitations associated with the fixed instruction fetch-execute model. They all have in common the

concept of a *configuration*: an amount of architecturally-visible instruction store that *separates the actions of fetch and execute*; after a fetch ("reconfiguration"), it can be persistent for some number of uses. 'Instruction' is used in a very general sense here, meaning anything that controls or affects the computation taking place. Because the fetch cost is amortized over many uses, a much larger, more detailed 'instruction' can be used. This detailed instruction in turn enables many performance enhancements depending on details of the reconfigurable architecture: unbound per-cycle ILP (instruction-level parallelism), tailored data width, fusion of successive operations, specialization of operations with constant inputs, etc.

This approach is beneficial only when the performance benefit outweighs the cost of fetching the large configuration. In simplified terms, it makes sense when

$$usesPerReconfiguration \times benefitPerUse > reconfigurationCost$$

This inequality assumes that a configuration may need to be loaded multiple times over a program's execution since it may be displaced by other configurations from a configuration state with limited storage capacity. The *benefitPerUse* value is relative to 'traditional' execution and includes any additional per-use overhead such as delay for data transfer. For many regions of the program, loading a configuration will not be beneficial: the region will not be used enough to overcome the configuration cost. The balance is also affected by how much benefit per use the region can gain from the more detailed 'instruction'; in fact some regions may derive no benefit even ignoring the configuration cost.

These considerations motivate hybrid architectures that combine a traditional microprocessor core with reconfigurable resources on a single-chip implementation [DeH94, RS94]. Different regions of a program are executed on either the microprocessor core or the reconfigurable coprocessor depending on their characteristics.

While hybrid microprocessor plus reconfigurable architectures have long been envisioned, for a long period they were simply not available commercially. During that time many research efforts built systems coupling a separate microprocessor with field-programmable gate array (FPGA) chips as the reconfigurable resource. FPGAs can be programmed to implement any digital circuit

up to a certain capacity, but the programmed circuit is not as fast or dense as if the circuit were built directly into silicon.

These systems typically fell into two categories: niche systems tuned for accelerating programs from a specific problem domain such as string matching or encryption; and proof-of-concept systems targeting general purpose computing. While the niche systems often showed significant speedup, the general purpose systems usually did not. They were successful as far as implementing parts of the program on the reconfigurable FPGA, but overall performance was typically disappointing, or must be assumed to be disappointing in many cases since it simply was not reported. Yet, such efforts were waging an uphill battle to achieve real performance benefits. The FPGAs were optimized for implementing 'random logic'—mainly working on one-bit values—rather than 32-bit operations typically used by software applications. The systems also suffered from high latency in communication between the FPGA and the microprocessor and/or the main memory system. Finally, since most commercial FPGAs were designed to be configured just once at power-up, the typical time for loading or changing a configuration was very large; this greatly impacted the configuration tradeoff inequality and greatly limited the fraction of a typical application that could benefit by using the reconfigurable resource.

While much effort was directed towards improving these proof-of-concept systems—by overcoming or exploiting certain quirks of the commercial FPGA being used—this work did not directly address the fundamental question of whether reconfigurable computing is an effective approach for improving performance of general purpose computation. This spurred the design of the Garp architecture.

1.2 Garp

The Garp architecture was designed to be a clean slate, best effort attempt at a hybrid architecture tuned towards improving the performance of general purpose applications. Combining a single-issue MIPS core, a reconfigurable coprocessor to be used as an accelerator, and a matched, high-performance memory system, Garp was assumed from the start to have a single-chip imple-

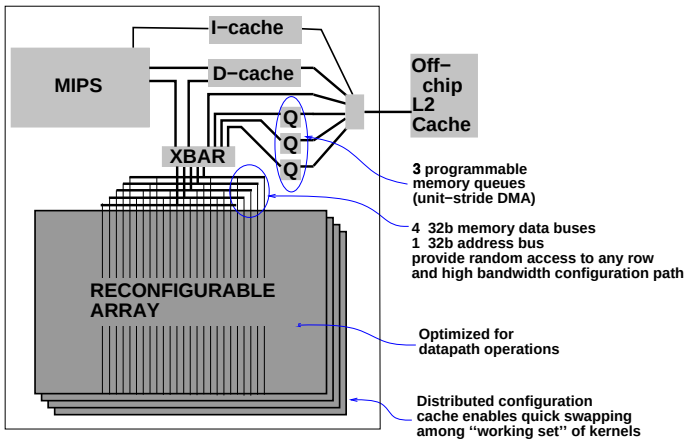


Figure 1.1: The Garp chip.

mentation (**Figure 1.1**).

Garp was designed with the intent that its reconfigurable hardware would accelerate loops of general-purpose programs. This goal led to the following design decisions for Garp, among the many discussed in Hauser's Ph.D. thesis [Hau00]:

- A few cycles of overhead for transferring data between processor registers and the reconfigurable hardware would be acceptable, since this overhead would occur only at the entrance and exit of loops. This is still much faster than would be possible using separate microprocessor and FPGA components.

- The reconfigurable hardware needed its own direct path to the processor's memory system, since most nontrivial loops operate over memory data structures. Relying on the main processor to shuffle data between the reconfigurable hardware and memory would be unacceptable; the processor would act as a bandwidth bottleneck and also add cycles of latency to every access.

- The reconfigurable hardware needed to be rapidly reconfigurable, since general-purpose applications tend to have many short-running loops.

- To facilitate binary compatibility, the array's timing is specified in terms of what computation and transfer delays could fit into a single array clock cycle, rather than in terms of absolute delays. This allows the same configuration to execute identically across different array implementations.

Garp was designed to fit into an ordinary processing environment that includes structured programs, subroutine libraries, context switches, virtual memory, and multiple users. It has the appropriate protection modes and instructions to support this environment.

1.3 Compiling C to Garp

For ease of programming hybrid systems, it is best if a single, software-like language is used for describing the entire application, encompassing computation on both the microprocessor and the FPGA. But traditional imperative software languages are basically sequential in nature; starting from there, it is a challenging task to exploit the reconfigurable hardware's parallel nature. Previous efforts have corrected this mismatch by using languages with constructs to explicitly specify either data parallelism [GS94, GG93] or more general parallelism [PL91, APR+, Wir98]. However, the more such a language's semantics deviate from those of sequential languages, the more difficult it is to train programmers to use it efficiently, and the more work is involved in porting "dusty deck" sequential code to it. The work presented in this thesis instead investigates automatic extraction and hardware acceleration[1] of code regions from standard sequential C code.

Automatic compilation of general purpose sequential code to hardware poses several challenges. In such code, basic blocks are typically small and contain little parallelism—a drawback considering that to fully exploit the reconfigurable array, the compiler must find and execute in parallel as many independent operations as possible. In addition, operations difficult to implement directly in hardware, such as subroutine calls, are often sprinkled throughout the code. Finally, loops often contain conditionals with rarely executed branches that interfere with optimization.

[1] Old habits die hard; this document will succomb to using the term "hardware" as in "mapping to hardware" or "hardware loop" when in fact the hardware being described is not "hard" at all since it is reconfigurable; it only shares the spatial form of compuation traditionally associated with hardware implementations.

Fortunately, researchers tackled similar challenges when building compilers for very-long-instruction-word (VLIW) machines. With slight adaptations, `garpcc` can use very much the same solutions, even though the Garp array's means of execution differs significantly from a VLIW processor's. The main construct borrowed from VLIW compilation is the *hyperblock* [MLC+92], a single-entry connected group of basic blocks. One hyperblock is formed for each loop to be accelerated.

The hyperblock exposes parallelism by merging included basic blocks, thereby removing scheduling boundaries. As part of the merging process, all control flow among the included basic blocks is converted to form of *predicated execution* where Boolean values originally controlling conditional branches are instead considered to be data values, used to enable operations and/or guide computation results. With `garpcc`'s approach, safe operations along all of the included control paths are executed unconditionally, and *select* operations are inserted at control flow merge points to select the correct results for use in subsequent computation (**Figure 1.2**). If a select operation's first operand (the predicate) is true, the result is the value of the second operand, otherwise the result is the value of the third operand. In hardware, a select operation is equivalent to a multiplexor ("mux"). Unsafe operations such as memory stores and hyperblock exits are guarded (enabled/disabled) by a predicate input. Predicates are expressions of the Boolean values that originally controlled conditional branches. The resulting effect is essentially speculative execution of all included paths; although this approach rapidly consumes resources, it also gives the best performance since it exposes the most parallelism and reduces critical path lengths. In addition, the computation in the hyperblock is transcribed to dataflow graph (DFG) form resembling static single assignment (SSA), making true dependences explicit and removing false dependences that artificially reduce ILP.

The hyperblock construct attacks the other problems—rare paths containing infeasible operations or other operations interfering with optimizations—by including only common paths, excluding others. Only the hyperblock is executed on the reconfigurable processor, while excluded paths are executed in software on the main processor. Without the exclusion ability, an infeasible operation on even an uncommon path would prevent any of a loop from being accelerated. Further-

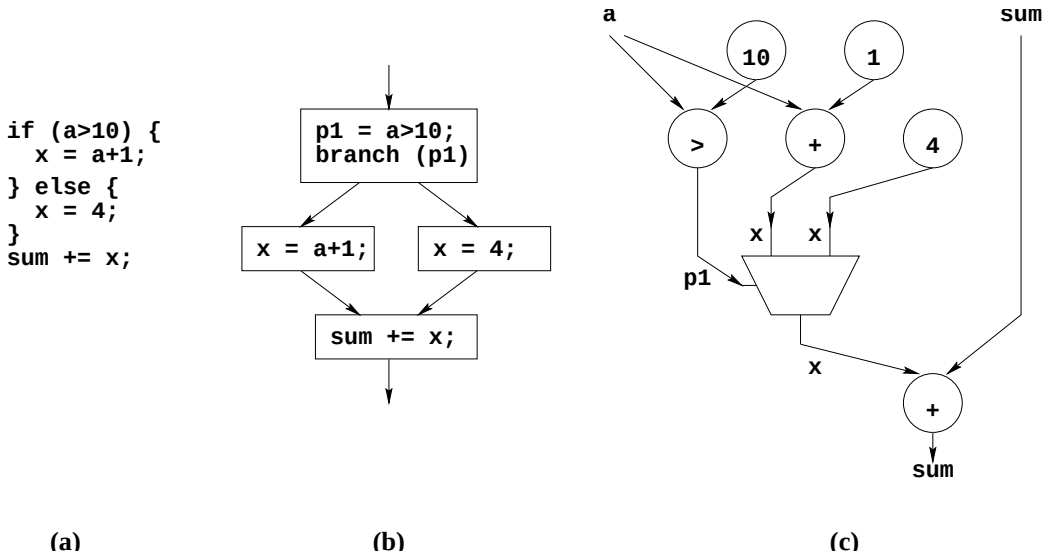


Figure 1.2: Predicated, speculative execution.

more, by excluding uncommon paths, the remaining paths typically execute more quickly because of eliminated dependence paths and increased optimization opportunities. Additionally, in the case of reconfigurable computing, less reconfigurable hardware resources are required. See (**Figure 1.3**). The exclusion of paths is a feasible approach owing to the close coupling of the Garp MIPS core and reconfigurable array; with more overhead, only extremely infrequent paths could be excluded.

Another challenge facing automatic compilation is construction of the configuration in time similar to that of software compilation. Original FPGA design flows based on bit-level Boolean logic were slow and also lost the regularity important for creating good datapath implementations. Given these drawbacks, previous compilation approaches migrated towards the use of modules—set patterns of configurations to implement a given multi-bit function using the FPGA's configurable logic blocks (CLBs). The drawback to this approach is that no optimization is performed across module boundaries, leading to configurations larger and slower than necessary.

`Garpcc`'s solution instead directly recognizes groups of operations that can be implemented by compound modules in the Garp array. The algorithm, based on instruction selection for complex instruction set processors, operates in linear time. The final datapath can viewed as a

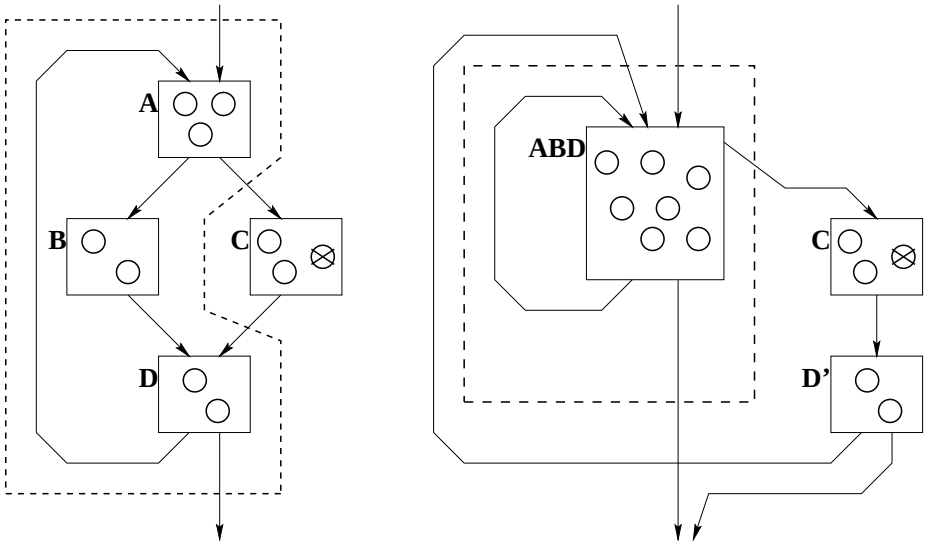


Figure 1.3: Hyperblock: exposing operation parallelism while excluding uncommon paths. When used by `garpcc`, only the hyperblock ABD is executed using reconfigurable hardware while other code is executed in software. Basic block D must be duplicated as D' since a hyperblock cannot be re-entered.

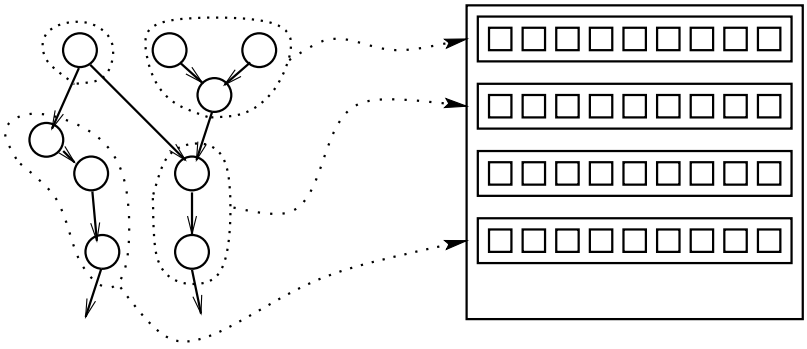


Figure 1.4: Mapping groups of operations to compound modules in the Garp array.

static, spatial implementation of a group of complex instructions interconnected by vertical buses (**Figure 1.4**). This approach grew out of experience hand-mapping computation to the Garp array; packing operations according to the fixed clock and utilizing the complex features available—particularly the memory accesses—made obvious the parallels with coding in assembly language. The algorithm also considers and determines relative placement of the compound modules, allowing simultaneous optimization of both computation and transfer delays.

1.4 Contributions

The contributions of this work are outlined below:

- The novel use of the hyperblock as a framework for hardware-software partitioning for a hybrid architecture is described. From each loop, one hyperblock—"kernel"—is formed, which may be the entire loop or just the subset consisting of the common paths. The kernel contains the computation eventually implemented in on the reconfigurable coprocessor when beneficial.

- A dataflow graph (DFG), constructed from the hyperblock using predication and speculation, is shown to be an effective representation for exposing instruction-level parallelism. The DFG is used not just as an auxiliary data structure to aid optimizations, but as the primary data

structure for each kernel. Furthermore, the DFG representation is shown to enable effective and efficient implementations of a wide class of analyses and optimizations, for example, finding uses for Garp's memory queues.

- "Pruning" is introduced as an approach for carving kernels out of loops both for fitting to available resources and for improving performance. The DFG structure is exploited to estimate the effect of each prune by suppressing those parts of the DFG that would be removed. Both iterative and enumerative approaches to pruning are investigated.

- A method is described for rapidly constructing efficient fully spatial datapaths on the reconfigurable coprocessor, grouping operations into optimized modules. Unlike simple module instantiation, it allows optimization across module boundaries; unlike traditional logic optimization at the single-bit gate level, it is fast, retains datapath regularity, and allows exploitation of special features of the coprocessor. Furthermore, the algorithm simultaneously considers relative placement of the modules when performing the grouping, which is important since these two tasks interact. It is also timing-sensitive in that it optimizes paths along the critical path/cycle. Finally, with some simple modifications this method is extended for constructing pipelined datapaths, further increasing parallelism and performance.

- A complete, robust compilation path from C to executable for the Garp architecture has been constructed, dubbed `garpcc`. Combined with a cycle-accurate simulation of a realistic implementation of Garp, this allows quantitative evaluation of the current Garp/`garpcc` system across a number of large benchmarks. Identification of weaknesses is mainly confined to `garpcc`, since until these are addressed, it is premature to fault Garp or its approach to reconfigurable computing in general.

Chapter 2

Compilation Flow Overview

This chapter outlines the compiler flow to provide context for the subsequent detailed description of Garp-specific compilation phases. First the complete start-to-finish compiler flow is described. Then an overview of the kernel extraction process is given. Some small but complete and real examples of compilation, including intermediate file printouts, are attached in Appendix A.

2.1 Overall compiler flow

The SUIF compiler [AAW+96] was selected as a starting point because of its modular construction and well-documented libraries. SUIF libraries provide a common interface to the intermediate representation (IR) in memory, plus means for reading and writing the IR to a standard file format. Typical SUIF compilation occurs as a sequence of passes, each of which is separate program that reads a SUIF file, modifies the IR in some way, then writes out a new SUIF file.

Garpcc compiler driver The top-level program `garpcc` is a Perl script that reads its command line options including names of input files, and based upon those values executes the appropriate sequence of other passes/programs with the appropriate arguments to achieve the desired compilation. The other programs include the C preprocessor, standard SUIF passes (included in the SUIF distribution), custom SUIF passes (written as part of this thesis work), custom Garp synthesis tools, other Perl scripts, and a slightly modified MIPS-targeting `gcc` compiler.

As with a standard C compiler, `garpcc` can compile the input source files either individually or as a group. The latter approach, supplying all source files simultaneously, allows for analysis and transformations across file boundaries, in particular pointer analysis and procedure inlining.

Front end processing The compiler's input is ISO (International Organization for Standardization) C [HS95]; the programmer is not expected to insert any hints or directives in the source code. Therefore SUIF can be used for the front-end phase of compilation—parsing and standard optimizations—with no modification.

As part of the SUIF flow, standard C preprocessing is performed by GNU cpp. Since `garpcc` is usually acting as a cross-compiler (running on a Solaris or Linux machine while compiling to the Garp MIPS-based platform), care must be taken that the correct set of system header files are used. Because some optional user directives may take the form of comments in the source code, the preprocessor may be instructed to not strip comments from the source file. In this case the parser includes the comments as annotations attached to instructions in the SUIF representation.

Initially the representation is "high" SUIF, which is structurally very close to the original C source code's abstract syntax tree, retaining the original nested structured control flow constructs—FOR loops, DO/WHILE loops, and IF/THEN/ELSEs. High SUIF also retains array address calculations as special array instructions rather than dismantling them to basic arithmetic on pointers.

Common optimizations are performed on the high SUIF, including loop-invariant code motion, dead code elimination, constant folding and propagation, and common subexpression elimination.

At this point procedure inlining is optionally performed. If so, `garpcc` reruns optimizations afterwards because inlining will likely have created more possibilities for optimization. Inlining must be applied intelligently or else undesirable effects may occur: code explosion, multiple identical kernels, etc. Inlining is performed in two situations:

- The call site occurs inside a loop, and the callee is a small, leaf, loopless procedure.
- The call site has at least one constant parameter, and the callee contains a loop. A more

sophisticated approach would trace the flow of the parameters within the callee so that the inlining is only performed when the constant parameter(s) are guaranteed to impact at least one loop in the callee.

Inlining guided by this simple heuristic proved to have limited benefit; in retrospect inlining should be performed after profiling. This is discussed further in Chapter 9.

Complete unrolling of small loops with small fixed number of iterations is selectively performed. This allows an outer loop to become the inner loop and be accelerated on the array. Although the original inner loop can no longer be accelerated by itself in such cases, it is likely not a good candidate anyway because of its low iteration count.

The next transformation applied, "`porky -loop-cond`", looks for loops containing an invariant conditional, and pulls it outside the loop, creating two specialized loops. A structure such as

```
d = ...;
for (...) {
    if (d<20) {
        xxx
    } else {
        yyy
    }
}
```

with '`d`' loop-invariant, is transformed to:

```
d = ...;
if (d<20) {
    for (...) {
        xxx
    }
} else {
    for (...) {
        yyy
    }
}
```

`Garpcc` then applies strength reduction to multiplication operations in FOR loops where one multiplicand is the loop index and the other multiplicand is a non-constant but loop-invariant expression. The FOR step amount can be a constant or a loop-invariant expression. This optimization

is important because the Garp compiler does not currently support variable $\times$ variable multiplication on the reconfigurable array. This strength reduction replaces such a multiplication with a simple addition. The algorithm used is similar to that described in the Dragon book [ASU86], Section 10.7, Algorithm 10.10, with the extension for loop-invariant expressions, although the case here recognizes only the loop index but not other induction variables.

Memory array dependence analysis is performed on the high SUIF representation. It is performed at this point because it relies on high level information such as FOR loop bounds and array access information. Dependence analysis results are attached as annotations to load and store instructions. These allow dependence information to be used at later points in compilation, even after FOR loops and array instructions have been dismantled. Specifically, this information will later be used to eliminate unnecessary ordering restrictions between memory accesses, to eliminate redundant loads and stores, and to help determine when it is legal to utilize Garp's memory queues.

`Garpcc` then dismantles the high-level structures of high SUIF, resulting in "low" SUIF. This is necessary for subsequent analysis and transformations that require a basic block, control flow graph representation of the computation. This pass decomposes FOR loops, WHILE loops, and IF/THEN/ELSEs into labels and conditional branches. It also decomposes "blocks" (lexical scopes) within a procedure; all variables become visible throughout the entire procedure. Variable renaming is performed as necessary to avoid name conflicts. This step also dismantles array instructions into explicit address arithmetic.

`Garpcc` then performs another set of optimizations on the low SUIF representation. Many of the same optimizations are performed again, and some new ones are applied. An example of a new optimization applied here is unstructured control flow optimization, which improves some inefficient branch sequences that may result from the dismantling step.

The SUIF representation at this point is viewed as a control flow graph (CFG) of basic blocks. Each basic block consists of a sequence of SUIF instructions, beginning with a label (the target of branches from other basic blocks) and ending with a branch instruction.

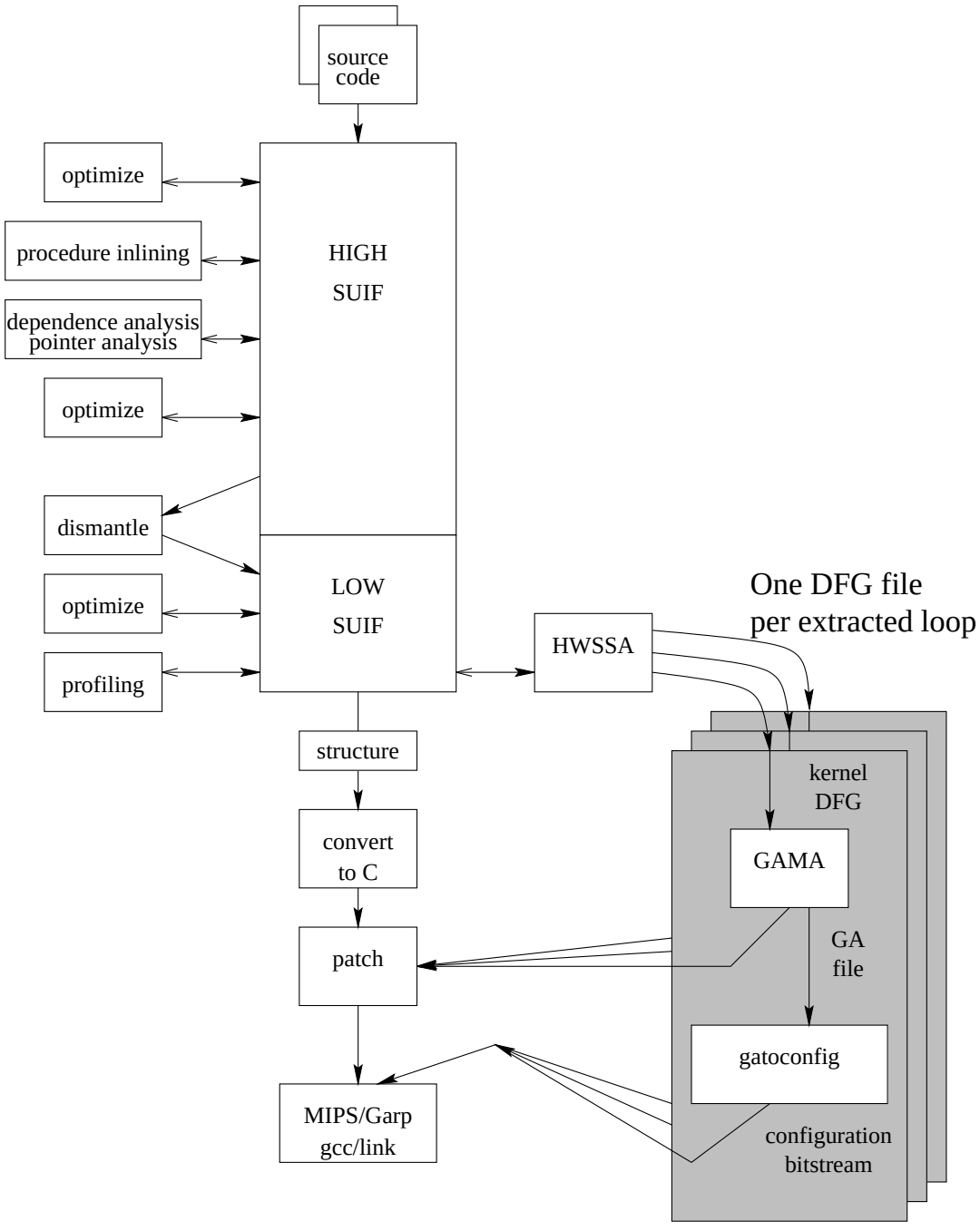


Figure 2.1: Garp Compiler Structure

Profiling Execution counts for each CFG block and edge are collected at this point. Each procedure entry and conditional branch is labeled with a unique number. Then a special version of the program is instrumented, compiled, and executed with a sample dataset. This instrumented version collects data regarding how many times each procedure is entered and how many times each conditional branch goes each direction, then writes out a file with this data. This data is correlated with the unique labels in the non-instrumented SUIF, and the counts are inserted as annotations for later use by the hardware loop extraction step.

Kernel extraction The first Garp-specific part of the compiler flow is the extraction of kernels for hardware acceleration. This is accomplished by a pass called HWSSA. The HWSSA pass reads in a low SUIF file. It writes out a modified SUIF file as well as an ASCII file representing the optimized dataflow graph (DFG) for each extracted kernel. An overview of kernel extraction is given in the next section.

Each DFG file will be fed to the short synthesis tool chain as described below. The modified SUIF file contains two versions of each loop from which a kernel is extracted. The original version of the loop is left intact and is now called the "software" version. The "hardware" version of the loop, at the end of the pass, contains in fact no loop, but instead basic blocks that contain the required information, in the form of annotations, required for interfacing with the extracted kernel. A "switch" basic block is inserted before the loops to select between the hardware and software versions. A subsequent step will hardwire the conditional branch in the switch block to instead be an unconditional branch to one or the other loop version.

HWSSA also writes a summary data file for use by the `garpcc` compiler driver; this file contains for example the name of each extracted kernel.

Hardware synthesis `Garpcc` then synthesizes a configuration bitstream for each dataflow graph. If the first attempt fails because it is too big or unroutable, synthesis is attempted again with more conservative options. If synthesis does not succeed after a small number of tries, the kernel is declared to have failed synthesis.

Each synthesis attempt invokes two tools. The first is GAMA, which reads in the ASCII

dataflow graph file, synthesizes a datapath, and writes it out as an ASCII GA (Garp array) file that *almost* completely specifies the configuration—the detailed function of each placed CLB as well as all routing within a row or between adjacent rows. The second tool is `gatoconfig`. Its main task is parsing the GA file and writing out the final encoded configuration bitstream in the form of a sequence of 32b integers, checking for conflicts in the GA file in the process. `Gatoconfig` also performs the only remaining assignment task—final assignment of vertical buses connecting modules in different rows.

HW/SW interfacing The software part of the program is converted from SUIF back to C. As part of this process, some attempt is made to reconstruct array accesses and structured control flow from the low SUIF, since this helps the subsequent final `gcc` C compilation recognize optimization opportunities. Annotations in the SUIF file are written out as comments in the generated C code. The C code is then scanned and modified (patched), combining information from the C comments with symbolic information in the GA files to insert the correct instructions for interfacing with the Garp array.

For an unsuccessful kernel, the C code is patched to jump to the software version of that loop. The hardware version of the kernel becomes unreachable and is eliminated.

For a successful kernel, the C code is patched to jump to the hardware version of that loop, and interface instructions as listed below are inserted as necessary (most kernels require only a subset, although pipelined execution is slightly more complex):

- The instruction that loads the correct configuration and clears all array registers. Garp is smart so that if the requested configuration is the same as the currently loaded configuration, a redundant load is not performed.
- Instructions to initialize the Garp memory queues.
- Instructions to move any live values to the coprocessor.
- The instruction to activate the coprocessor.

- Instructions to determine which exit was taken and then branch to the correct continuation point in software (only if there is more than one exit from the kernel).
- Instructions specific to that exit to move the appropriate live values from the array.
- Instructions to flush the Garp memory queues; required only if a queue store was utilized.

All of these tasks require the use of instructions unique to Garp's instruction set. Fortunately `gcc`'s "`asm`" directive provides extremely facile access to these instructions, transparently interfacing source-level variables to assembly-level instructions.

For each successful kernel, an integer array declaration is inserted at file-global scope. The configuration data output from `gatoconfig` is `#include`'d into the C source as the initialization data for that array.

Discussion of HW/SW interfacing in any more detail benefits greatly from a real example, and thus is deferred (Appendix A).

Final compilation The modified C code is then compiled by `gcc` resulting in the object code that runs on Garp's MIPS core. This version of `gcc` has been slightly modified to recognize the new Garp instruction mnemonics when they appear in ASM directives. Thus the new Garp instructions added to the C code by the patch step get translated directly to instructions in the object code, while `gcc` compiles the surrounding C code to standard MIPS object code. The final link step is not modified.

2.2 Kernel extraction overview

Figure 2.2 shows the flow for kernel extraction (hyperblock formation). Although the framework of kernel extraction is built on the concept of the hyperblock, `garpcc`'s process of kernel formation differs from the original VLIW hyperblock formulation [MLC+92] in a number of ways.

- The hyperblock formation here is split into two phases—first, `garpcc` eliminates paths with operations that are infeasible for hardware implementation; then later, it utilizes profiling data

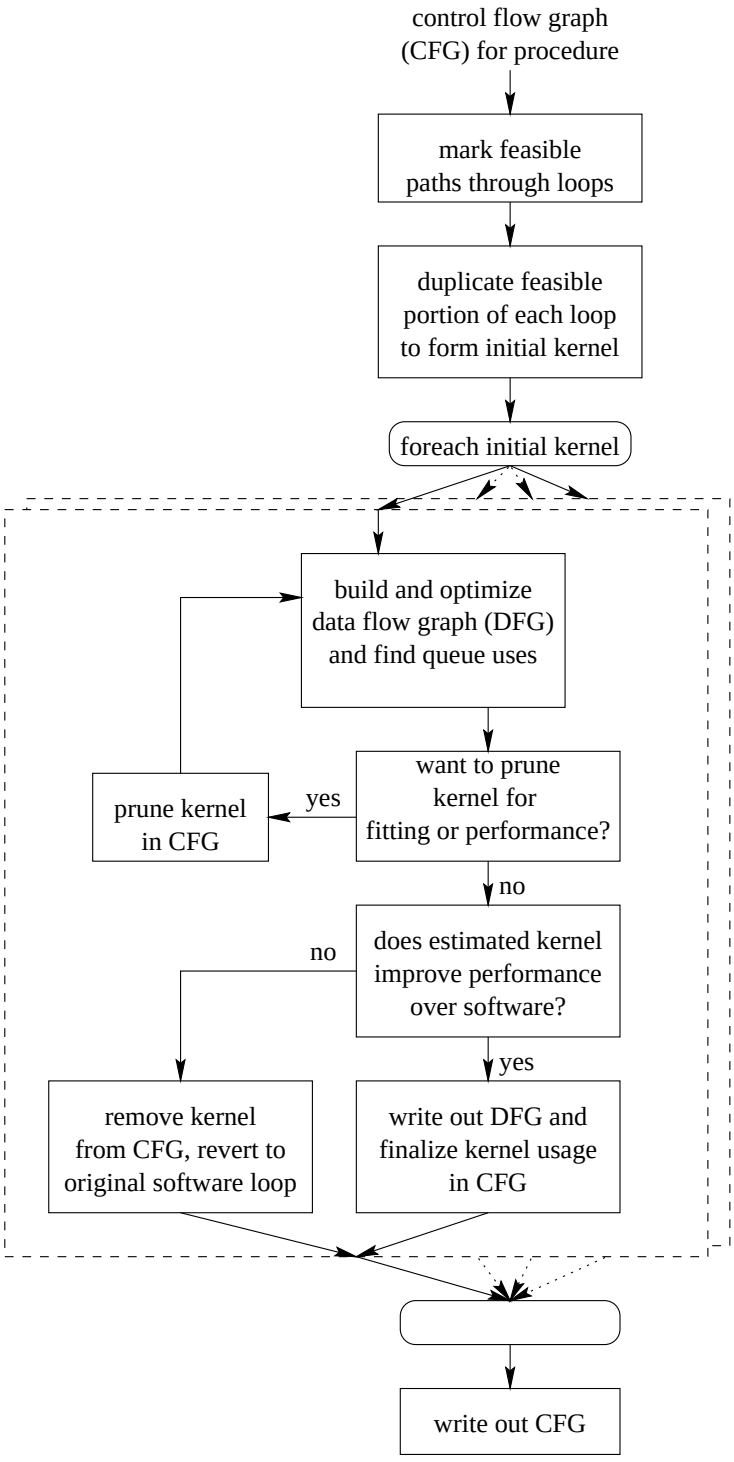


Figure 2.2: Kernel extraction flow

and hardware estimation when it further eliminates paths with the goals of fitting to available resources and/or improving performance. Although the VLIW hyperblock formation heuristic described in [MLC+92] accounts for "hazardous" operations, they are not strictly prohibited and are instead weighted into the overall heuristic.

- The hyperblock/kernel formation here uses a subtractive rather than additive process. From the duplicated loop of all feasible paths, additional paths are removed for fitting and/or performance improvement. At this later stage the DFG representing the compuation has already been built; it is used to assist in evaluation of the costs and benefits of eliminating a given path or paths.

- As a small technical detail, tail duplication is reversed. In both cases tails are required because hyperblocks cannot be re-entered, so instead the tail, a copy of one or more blocks in the hyperblock, is executed. In the VLIW formulation, the hyperblock is formed from the original blocks while tails are formed from new copies. In the formulation here, the hyperblock is carved from a copy of the loop, while the original loop (implemented in software) provides tails as necessary downstream from hyperblock/kernel exits.

2.3 Thesis overview

The tasks involved with kernel formation are detailed in the next few chapters. Chapter 3 discusses initial kernel formation, removing just infeasible operations from kernel candidates, which are copies of the original loops. Chapter 4 discusses DFG construction and optimization, while Chapter 5 covers how potential uses of Garp's memory queues are found and exploited. Pruning for fitting and performance is covered in Chapter 6.

The remainder of the thesis is organized as follows. Chapter 7 describes the synthesis phase that occurs after kernel finalization and Chapter 8 describes the synthesis extension for pipelined execution. Although some results are presented in other chapters, Chapter 9 presents experiments that bring everything together. Chapter 10 concludes with a summary, an overview of related work, and future directions for this work.

Chapter 3

Initial Kernel Formation

Original loops in the program may contain infeasible operations such as library calls that cannot be executed directly by the Garp array (the current set of infeasible is described later). There is no way these can be implemented on the Garp array, so as a first step is to form an initial kernel containing just the paths in the original loop containing no infeasible opeartions.

For each loop, feasible paths are marked and then duplicated to form the initial hardware kernel for that loop. There may be no feasible paths through the loop, in which case no initial kernel is formed. Even if an initial hardware kernel is formed at this point, there are a number of reasons why it may later be rejected for hardware acceleration.

The subsequent pruning step (Section 6) may remove some of the feasible paths from the initial kernel in order to allow the kernel to fit into available resources or to achieve higher performance.

3.1 Background: the CFG and natural loops

The first step in hardware kernel extraction is finding loops in the original program. Rather than relying on loop constructs in the original source code, the compiler utilizes control flow graph analysis to recognize loops. This allows the extraction and acceleration of a more general class of loops, including even those formed by backwards `goto` statements.

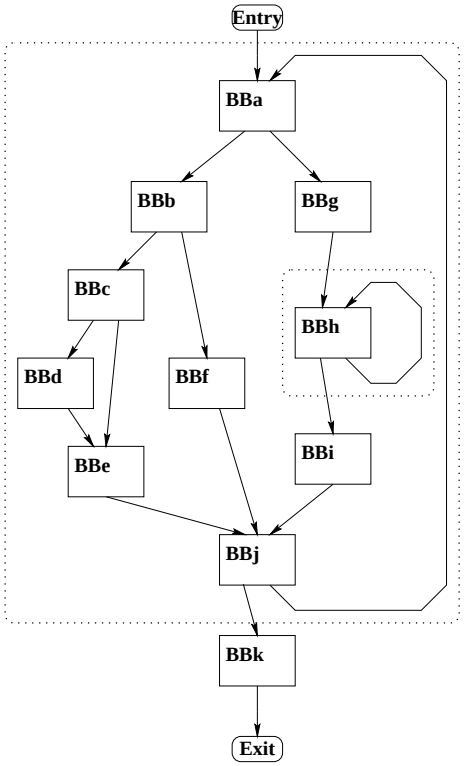
The compiler breaks up each procedure into basic blocks, which are instruction sequences with no branches into or out of the middle. At the end of each basic block is a branch that controls which block executes next. These branches connect all the basic blocks of a procedure into one control flow graph (CFG). Thus the basic blocks are the nodes of the graph, and possible branches from one block to another form directed edges between the nodes. Each multiway branch (originating from a SWITCH/CASE statement) has been dismantled to an equivalent set of conditional branches, so each basic block has at most two successors. The CFG also contains special Entry and Exit blocks that contain no instructions.

Analysis of the CFG allows automatic recognition of loops. The discovery of *natural loops* is based on the concept of *dominators* [ASU86]; these and related concepts are reviewed below and illustrated in **Figure 3.1**.

Dominators A basic block X *dominates* basic block Y if and only if every directed path from the Entry block to basic block Y goes through basic block X. Domination is reflexive, so that in all cases X dominates itself. Calculation of dominators is described in [ASU86].

Post-dominators The definition of post-dominators is analogous to that of dominators. A basic block X *post-dominates* basic block Y if and only if every directed path from basic block Y to the Exit block goes through basic block X. Postdomination is also reflexive.

Natural Loops and Backedges A formal definition of a loop is based on the definition of dominators. A basic block L is a *loop entry* if and only if there exists at least one basic block Y such that (i) L dominates Y, and (ii) there is an edge from Y to L. The edge from Y to L is called a *backedge*. L and Y can be the same. The loop defined by loop entry L is the set of basic blocks $\mathcal{L}_L$ such that for each basic block $X \in \mathcal{L}_L$, (i) L dominates X, and (ii) there is a directed path from X to L such that every basic block in the path is dominated by L. A loop may have multiple backedges. Natural loops may be nested, so that a basic block may belong to multiple nested natural loops.



Domination relation:
- BBa dom all others
- BBb dom BBb, BBc, BBd, BBe, BBf
- BBc dom BBc, BBd, BBe
- BBd dom BBd
- BBe dom BBe
- BBf dom BBf
- BBg dom BBg, BBh, BBi
- BBh dom BBh, BBi
- BBi dom BBi
- BBj dom BBj, BBk
- BBk dom BBk

Loop Entries:
- BBa (backedge BBj->BBa)
- BBh (backedge BBh->BBh)

Figure 3.1: Dominators and Natural Loops.

Irreducible Loops It is possible for there to be a cyclic path in a CFG that does not fit the definition of natural loop given above. Such a cycle is termed an irreducible loop. An irreducible loop typically occurs when there are multiple entry points to the cycle; in this case, no basic block dominates all of the others in the cycle, and thus there is no well-defined loop entry or backedge.

Irreducible loops occur very rarely in hand-written C programs. Irreducible loops can always be eliminated through the duplication of basic blocks; however, such transformations were not implemented in `garpcc` because of the rarity of irreducible loops.

3.2 Marking feasible paths in loops

This phase marks feasible paths through each loop, determining the *hwloopid* attribute for each basic block in the process. The value of the hwloopid attribute for a basic block refers the loop entry block of the kernel to which it belongs, or is NULL if the basic block belongs to no kernel.

The algorithm operates over a procedure's entire CFG, which may contain several individual (possibly nested) loops, which could each spawn a kernel. The algorithm to set the hwloopid attribute for every block is performed in two passes as described in the following text; the pseudocode is shown in **Figure 3.2**.

The first pass progresses forward through the CFG. A feasible loop entry block is marked as belonging to its own kernel (i.e. its hwloopid is itself); this acts as a seed from which feasible paths grow forward. The hwloopid is propagated from a marked block to each feasible successor that is dominated by that hwloopid. For any infeasible successor, its hwloopid is set to NULL, thus killing any paths through it.

The second pass is a backwards trimming phase. It sets to NULL the hwloopid attribute of each block that does not have any successor sharing the same hwloopid. This works to unmark any block not having a directed path back to the loop entry indicated by its hwloopid attribute (avoiding other loop entry blocks). Note that this second phase will completely unmark any loop that has no feasible paths through it.

Nested loops are not directly supported. Therefore a nested loop entry is infeasible with

```
//— Forward pass
N = blocks in CFG in reverse postorder;
foreach block n in N {
    n.hwloopid = NULL;
    if (infeasible(n)) continue;
    if is_loop_entry(n) {
        n.hwloopid = n;
    } else {
        foreach block p in preds(n) {
            if (p.hwloopid AND p.hwloopid dominates n) {
                n.hwloopid = p.hwloopid;
            }
        }
    }
    //— Check for irreducible loop
    if (n.hwloopid) {
        if (n has successor s such that
            s is not loop entry n.hwloopid AND s.hwloopid == n.hwloopid) {
                //— Found bad cycle; unset hwloopid
                n.hwloopid = NULL;
        }
    }
}
//— Backwards pass
N = blocks in CFG in reverse topological order;
foreach block n in N {
    if (n.hwloopid == NULL) continue;
    selected_successor = FALSE;
    foreach block s in succs(n) {
        if (s.hwloopid == n.hwloopid) {
            selected_successor = TRUE;
        }
    }
    if ( !selected_successor ) {
        n.hwloopid = NULL;
    }
}
```

Figure 3.2: Algorithm: Hardware Loop Selection and Trimming

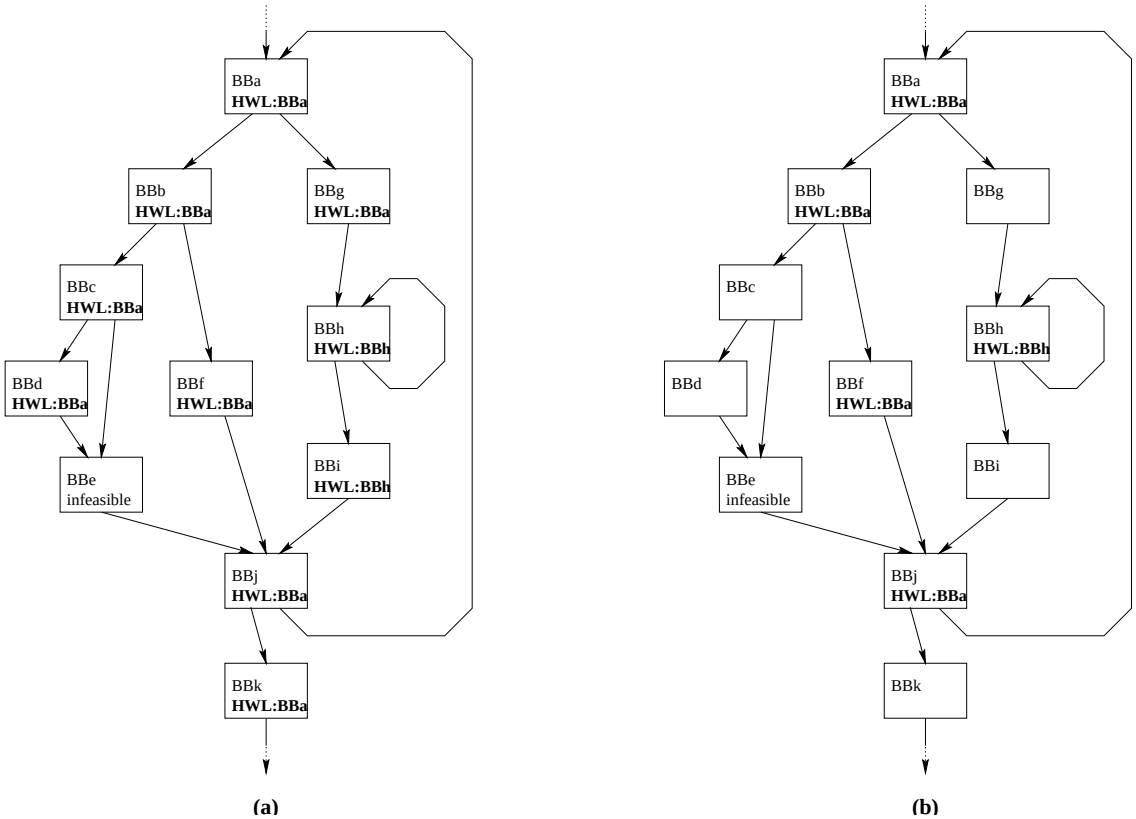


Figure 3.3: Marking feasible paths in loops: (a) after forward pass, (b) after backwards pass, which unmarks BBc, BBd, BBg, BBi, and BBk.

respect to an outer loop. However such a loop entry is the start of a new kernel. Thus although a basic block can belong to multiple natural loops, it can belong to at most one kernel: that of the innermost loop containing it. Yet both inner and outer loops can become individual kernels as illustrated below.

Figure 3.3 shows an example with nested loops. Basic block *BBj* has two marked predecessors, but they are marked with different hwloopid attributes; *BBj*'s hwloopid attribute is set to *BBa* since only *BBa* dominates *BBj*. A basic block will never have predecessors with two different hwloopids that both dominate the block.

Further considering **Figure 3.3**, after the backwards pass, basic blocks *BBg* and *BBi* are unmarked even though they are part of the natural loop headed by *BBa*. That is because all paths

through them also go through the inner loop headed by *BBh*.

Still considering **Figure 3.3**, *BBe* is infeasible. On the forward pass, *BBc* and *BBd* are marked as belonging to *BBa* while *BBe* does not have its hwloopid set because it is infeasible. During the backwards pass, however, *BBd* and then *BBc* have their hwloopid values cleared since they do not have a successor belonging to the same kernel. This is how the compiler resets blocks that are feasible but have no feasible path to a backedge back to the kernel entry.

Care is taken to recognize irreducible loops that occur inside of natural loops. Such a cycle would not be handled correctly. Thus part of each irreducible loop must be eliminated. It is not necessary that every block in the cycle be treated as infeasible, as long as the final set of marked blocks does not contain an illegal cycle. The block(s) that are unmarked are essentially chosen randomly, depending on the exact order that the blocks are visited. This may lead to a suboptimal decision in some cases, since a heavily executed block may be eliminated rather than a rarely executed block. Further effort was not devoted, however, because irreducible loops are rare, and an outer loop worth accelerating that contains an irreducible loop is even more rare. Irreducible loops occurring outside of natural loops are simply ignored by the algorithm, since there is no loop entry acting as the kernel seed.

If an outer loop has one or more paths through its body that avoid all of its inner loops, those paths will be marked as belonging to the outer loop. However, if no such path exists, the outer loop will be completely unmarked during the second phase. If compiler command line options indicate, an outer loop will be eliminated from HW consideration automatically even if it contains one or more paths avoiding all inner loops. This is typically a smart strategy when profiling information is not available.

The types of infeasible blocks are listed below.

Hardware-Infeasible Operations The first type of infeasible block is a *hardware-infeasible* basic block—one that contains an operation that cannot be synthesized in the reconfigurable array. The set of infeasible operations depends on the capability of the back-end tools for the target platform. At the time of writing, the list of infeasible operations for the Garp architecture and tool set is:

- Subroutine calls
- Floating point arithmetic
- Operations on 64bit data values
- Division or remainder, unless the operands are unsigned and the second operand is a constant power of two, in which case the operation is replaced by the equivalent shift or mask
- Compiler builtin functions that SUIF treats as special instructions. An example is `__alignof__(`*type*`)`. These are so rare that no effort was devoted to either attempting compile-time evaluation to a constant or attempting to represent the operation as a DFG node.

Note that a return can never occur inside a natural loop. The basic block that ends with the return is actually outside of the loop, even if textually it occurs within the loop in the source code. So although return statements would be infeasible operations, `garpcc` need not look for them inside the loop.

Inner Loops As mentioned before, an inner loop entry is treated as infeasible from the point of view of an outer loop; it is the starting point for a new kernel.

User Annotations Finally, the compiler gives users the ability to manually declare that a basic block should not be implemented in hardware. A comment containing the string “no garp hw” located anywhere in the basic block will cause the basic block to be marked as infeasible. This facility can be used to inhibit an entire loop or to exclude specific paths from a kernel. No such annotations were added for any results presented this thesis.

3.3 Loop duplication

At this point the basic blocks used in the formation of each initial kernel have been marked. As will be seen, it is convenient for the compiler to keep a complete “software” ver-

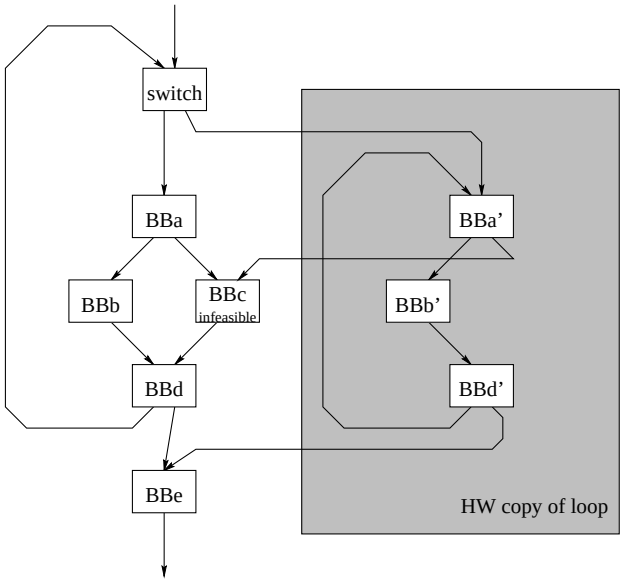


Figure 3.4: Loop Duplication

sion of each loop, and make a "hardware" copy from which the hardware implementation will be formed.

Consider **Figure 3.4**. *BBc* is infeasible, so only *BBa*, *BBb*, and *BBd* will be marked with hwloopid *BBa*. Each of these blocks is duplicated to form *BBa'*, *BBb'*, and *BBd'*. The hwloopid attribute in the hardware loop is adjusted to point to the new duplicated loop entry block *BBa'*, while the hwloopid attribute for each blocks in the original loop is reset to NULL.

An edge between two blocks with the same hwloopid *BBa →BBb* results in a corresponding edges between the copies of those blocks *BBa' →BBb'*. An edge from a block *BBa* with a hwloopid to a block *BBc* with no hwloopid or a different hwloopid results in an edge from the duplicated block *BBa'* to the original successor *BBc*. This edge *BBa' →BBc* is a *kernel exit*, and *BBa'* is termed an *exit block*. Kernel exits result from natural loop exits (*BBd' →BBe*), infeasible paths (*BBa' →BBc*), or from additional pruning of rarely-executed paths (Chapter 6).

A switch block is added to the CFG for each duplicated loop. The switch block has a conditional branch, with one outgoing edge to the original loop entry and one outgoing edge to the

new version's loop entry. Ultimately, this branch will be replaced with an unconditional branch to one of the loop versions. All edges originally entering the loop entry block from outside the loop are redirected to the switch block. Furthermore, all backedges in the software version of the loop are redirected from the original loop entry to the switch block; this allows the kernel to be reentered at the start of the next iteration after an excluded path exit. Back edges in the hardware loop remain pointing to the hardware loop entry.

Introduction of Irreducible Loops When a kernel has excluded paths, duplication results in extra entrances to the original software version of the loop. This changes the original software loop into an irreducible loop. This would cause concern, since irreducible loops are typically not subject to optimization and would get worse performance. However, it turns out that this is not a concern in either of the two eventual cases.

Looking ahead, the conditional branch of the switch block is ultimately replaced with an unconditional branch, depending on whether any of the kernel is implemented in hardware. The two cases are described below:

- If any of subset of the loop remains for acceleration in hardware, the switch block points to the hardware loop entry. In this case, the basic blocks in the SW loop provide 'tails' as in those created through tail duplication in hyperblock formation [MLC$^+$92]. In both cases, once an exit from the hyperblock/kernel is taken, the remainder of the iteration must be performed in the tail. In this case the computation in the tail is in fact acyclic and should not be considered for loop-based optimization.

  When an iteration exits hardware to execute an excluded path in software, the software loop backedge returns to the switch block, which in turn re-enters the hardware loop for the remaining iterations.

- If none of the loop is implemented in hardware, the switch block transfers control to the software loop. The entire hardware copy of the loop is ultimately removed since it is unreachable; this also removes side entrances to the software loop. The switch block itself is

optimized away since it does nothing. Thus the loop reverts exactly to the original software natural loop and is subject to optimization.

3.4 Invariants

The initial kernel formation algorithms guarantee the following ‘golden’ invariant:

Every hw block is part of one or more cycles of hw blocks, all of which include the hw loop entry.

These invariants follow from the golden invariant:

Every hw block has a hw successor.

Every hw block that has a sw successor has two successors: the sw successor and a hw successor. Such a block is called an exit block.

All of these invariants are preserved by the subsequent pruning phase (Chapter 6).

3.5 Profiling Count Adjustment

For basic blocks that are duplicated, most iterations will use the hardware version of the block’s computation, but some iterations might execute the software version (in the tail after kernel exit). Therefore, `garpcc` reallocates the original profiling count for a basic block that is duplicated between the two copies.

The loop starts every iteration in the hardware loop; therefore the hardware loop entry block has the same profiling count as the original loop entry. However, once any path leaves the hardware loop, it does not re-enter within the same iteration. Therefore counts downstream will be reduced proportionally. More exactly, counts on blocks downstream from an eliminated re-entry point will be reduced.

Calculating the adjusted profiling counts is straightforward. The entry block is given the original profiling count for that block; its outgoing edges also receive the original counts. Then every block in the hyperblock is visited in topological order. First, its block count is calculated as the sum of the counts on incoming edges (which could only originate within the hyperblock).

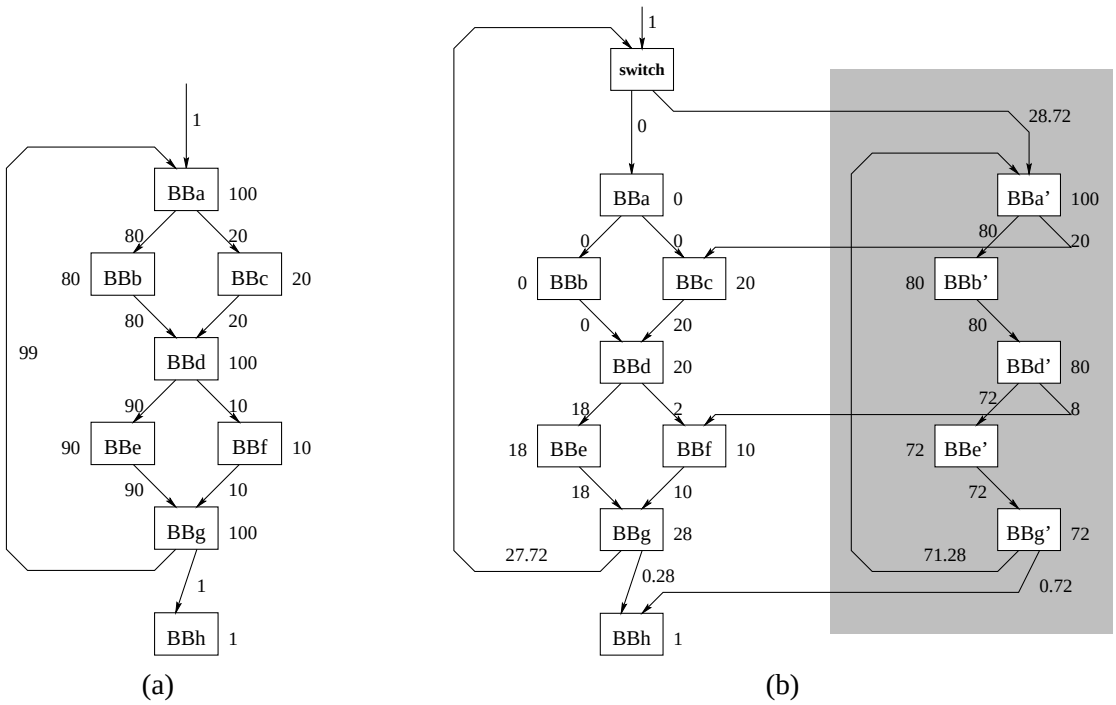


Figure 3.5: Profiling count adjustment. (a) original loop and profiling data, (b) adjusted hardware counts, and corresponding software tail execution counts.

Next, counts on the outgoing edge(s) are calculated by splitting the block count in proportion to the original counts on the edge(s).

The number of executions expected in the software tail duplicate version of each block can then be calculated. It is simply the difference between the original count and the adjusted hardware count.

Figure 3.5 shows an example with two exits. In the hardware adjusted counts, *BBd'* has a count of just 80 compared to the original 100; that is becuase the 20 iterations that branch to *BBa'* →*BBc* continue to *BBd* in the software tail rather than returning to *BBd'* in the kernel. The outgoing edges from *BBd'* are still proportioned the same—90% versus 10%—but are scaled down to match the new block count of *BBd'*. This assumes no correlation between successive conditional branches; path profiling [You98, BL96] would provide more accurate estimates when there is correlation, but it was not utilized.

Counts are expressed as floating point values. While there cannot truly be a fractional execution of a control path, the non-integer values express a probability combined with original counts. For example, assuming a random distribution, the very last iteration of the loop has a 72% chance of remaining in the kernel until *BBg'* versus a cumulative 28% chance it will have taken an earlier exit. In other words, there is no way to know for sure whether the final iteration will be one of the 72 out of 100 that pass through *BBg'* or one of the 28 out of 100 that does not. Thus the expected execution count for edge *BBg'* →*BBh* is 0.72. Profiling counts will be used in heuristics to calculate "expected" costs averaged over all executions, so non-integer values cause no problems.

Chapter 4

The Data Flow Graph (DFG)

After the blocks in the hyperblock have been selected, HWSSA builds a dataflow graph (DFG) to represent the computation. The DFG can be considered to be a stepping stone between the original software specification and the final hardware (spatial) implementation. HWSSA performs many important tasks in building the DFG: (i) Control dependence within the kernel is converted to data dependence: conditional branches within the kernel are eliminated through the introduction of predicates. The only remaining conditional branches are exits out of the kernel. (ii) Data producer–consumer relationships are made explicit via data edges in the graph; also, since a new DFG node is created for each definition, variable renaming is effectively performed, eliminating false dependences. These effects are similar to static single assignment form [CFR+91] and lead to similar benefits. (iii) Any remaining ordering constraints between individual operations are also made explicit through additional edges.

These actions convert the sequential ordering of instructions to a partial order of DFG nodes, exposing parallelism. In addition, maximal control speculation is employed so that all safe operations execute every iteration, removing dependences between predicate calculations and those operations, breaking critical paths and further increasing operation parallelism.

Many optimizations are performed during DFG construction and as separate passes over the completed DFG. Because operations on paths excluded from the hyperblock do not interfere with optimization, the opportunities for optimization are often increased compared to optimiza-

tions operating over the entire loop, just as in the case of superblock / hyperblock compilation for VLIW processors [Mah92, Mah96]. The implementation of these optimizations is typically simplified when recast to the DFG representation since control flow has been eliminated and producer–consumer relationships are explicit. Finally, the fully-speculative approach utilized affects some of the tradeoffs involved in the optimizations.

The DFG is both a compiler memory data structure and a file format. The DFG is built and optimized in memory in a single pass that exploits the coexisting presence of the CFG and related SUIF intermediate representation. At the end of this pass, the DFG file is written out; this file contains all information required for correct synthesis.

The DFG consists of nodes connected by edges, the types of which are described in the following sections. The DFG is not hierarchical. The DFG is almost always cyclic. The DFG is target independent, but during DFG optimization, target-dependent estimates are utilized to evaluate tradeoffs.

4.1 DFG nodes

The nodes represent operations (computational, memory, or exit), kernel-invariant inputs, constants, or delays elements. The current implementation assumes that each node has no more than one distinct data output. A node with output may have fanout of any degree.

The node types are described below:

- **Computational Operator Nodes** are typical unary and binary computational operations having no side effects, and their output each iteration depends only on their inputs that iteration. Operator nodes commonly result from a direct translation of a SUIF instruction. They can also result from the introduction of predicates by the DFG construction algorithm. The node holds an opcode as well as a type field that determines the type of its output value. Allowable types are the standard C integer types signed and unsigned `int`, `short`, and `char` (32, 16, and 8 bits respectively), as well as the non-standard Boolean type, which is treated as a 1-bit unsigned integer.

- **Memory Access Nodes** provide access to the same view of memory as software memory accesses, also sharing the same pointer encoding. A store takes an address input and a data input, producing no output. A load has an address input and a data output. Supported data sizes are 8, 16, and 32 bits. There are predicated versions of both load and store; these have an additional predicate input. In Chapter 5, queue access nodes will be introduced, which need no address input since they are instead initialized with a starting address. Queue access nodes can have any of the three standard data sizes mentioned above, and can be predicated or not. Both regular and queue loads can be executed speculatively, simply meaning that a load may be attempted from any address without a fatal exception occurring; if the address is invalid, the data returned will be a random value.

- **Constant Nodes** each output a value known at compile time. This value is included on the node.

- **Input Nodes** each hold a value not known at compile time but which is constant through the kernel execution (i.e. is kernel-invariant). The value is initialized before kernel execution starts and does not change. A special case is the memory address of a scalar, aggregate, or array variable. While addresses of global variables are in fact known at compile time, it is only at link time, and the linker does not know how to patch Garp configurations. Thus all addresses of variables are provided as Input nodes.

- **Hold Nodes** exist for (kernel)loop-carried variables. When a Hold node is inserted between a producing and a consuming DFG node, the consuming node will receive the value produced by the producer during the previous iteration. Multiple Hold nodes can be chained to achieve data transfer across multiple iterations. If there are N Hold nodes between a producer and consumer, it is equivalent to having a data edge with distance N.

- **Exit Nodes** halt computation in the kernel when their operand is true. There are predicated and unpredicated versions.

4.2 DFG edges

Edges indicate data producer-consumer relationships, ordering constraints, or liveness, as described below:

- **Data Edges** indicate a transfer of data between nodes. Because the datum must be produced before it is used, data edges impose an ordering between the execution of the two nodes.

- **Precedence Edges** indicate an ordering restriction between two nodes other than direct data producer/consumer. Precedence edges are required between a pair of memory accesses that might access the same memory location, unless both are loads. Precedence edges are also required between an exit and a store. Reordering of such a pair would result in a store being performed when it should not be, or vice versa.

  Precedence edges have a *distance* attribute indicating how many iteration boundaries they cross. In almost all cases the distance is 0 or 1. A precedence edge from node *nA* to node *nB* with distance d indicates that *nB* in iteration $i + d$ must be scheduled before *nA* is executed in iteration i. This allows uniform treatment of intra- and inter-iteration edges.

  The distance attribute on precedence edges between memory access nodes should not be confused with distances as in distance vectors [BGS94, Wol89] used in array dependence analysis. Here the precedence edge distance d is simply a conservative scheduling constraint passed to the synthesis back end. It does not guarantee that two accesses separated by d iterations definitely access the same location. Thus a precedence edge with distance 0 is equivalent to a '$\leq$' dependence, and a precedence edge with distance 1 is equivalent to a '$<$' dependence, using the terminology of [BGS94]. When stronger information about exact dependence distance is available, it is recorded as a separate annotation on the precedence edge. This stronger information is available only during DFG construction and optimization but is not written out to the DFG file.

- **Liveness Edges** only go to Exit nodes. They indicate the set of values that are live at that kernel exit and thus must be copied out of the kernel. Each liveness edge is annotated with

the name of the variable, since in general the variable cannot be deduced from the source node. These edges are necessary because the set of live variables to be transferred in general differs at each exit. Furthermore, the source DFG node for a given variable can be different at different exits.

Like precedence edges, liveness edges indicate an ordering. This ensures that all required live values have been calculated at the point that an Exit is taken. In Chapter 8 it will be seen that these edges are particularly useful when synthesizing pipelined datapaths, since they are further used to ensure that the variable's version from the correct iteration is available at the time an exit is taken.

4.3 DFG construction overview

The algorithm for building the DFG is described in Figure **4.1**. It performs a single forward pass, visiting each basic block in the kernel. This forward pass builds all of the DFG nodes, including DFG nodes directly translated from instructions as well as predicate calculation nodes and mux nodes added to implement predicated execution. The forward pass also builds all data and precedence edges contained within the iteration.

After the forward pass, a separate phase builds inter-iteration (loop-carried) data and precedence edges. Liveness edges are then added. Finally, many cleanup and optimization passes are performed on the DFG.

The following sections describe each aspect of DFG construction:

- Control relations between basic blocks
- Predicates
- Initial values
- Most recent definition of a variable
- Scalar variables in memory (register promotion)

```
build_dfg(hyperblock) {
    L = list of basic blocks selected for hyperblock, in topological order
    for each basic block B in L {
        if B is the hyperblock entry block
            predicate[B] = NULL;
        else
            predicate[B] = build_OR (predicates on B's incoming CFG edges);
        lastDefs = merge_incoming_defs(B);
        for each instruction I in B {
            build DFG node N corresponding to I;
            build N's incoming data edges;
            build N's incoming precedence edges;
            if N is not speculative, and predicate[B] exists,
                attach predicate[B] to N;
            if N defines a variable, update lastDefs list;
        }
        save lastDefsForBlock[B] = lastDefs;
        for each outgoing CFG edge E from B {
            predicate[E] = build_AND (predicate[B], local condition for E);
        }
    }
    build loop-carried data and precedence edges;
    find exit live variables at each exit & create liveness edges;
    remove false precedence edges;
    optimize DFG;
    remove redundant precedence edges;
    write out DFG;
}
```

Figure 4.1: Algorithm: Building the DFG

- Processing a basic block
- Merging data values: mux insertion
- Building precedence edges
- Forming loop-carried data edges
- Forming loop-carried precedence edges
- Live variables at exits
- Miscellaneous DFG optimizations
- Memory access optimizations
- Removal of redundant precedence edges

For readers with backgrounds in similar areas, the ideas will be familiar with perhaps slight modifications. Readers with background may also find this section overly detailed, yet may find the detail convenient when interested in how a particular aspect was handled. For readers with little or moderate background, this section may serve as a tutorial. There are many cyclic dependencies among the concepts in the different sections. Thus such readers should expect to have some unresolved questions while reading this chapter no matter what order is chosen. Hopefully all questions will be sufficiently answered after all has been read.

4.4 Control relations between basic blocks

In building and optimizing the DFG, it is useful to have knowledge of control relations between basic blocks—whether two basic blocks *always* execute the same iteration, whether they *never* execute the same iterations, or whether the execution of one implies the execution of the other in any given iteration. These are related to the ideas of domination/post-domination, but they are not exactly the same. Domination/post-domination act on the procedure scope, while here the concern

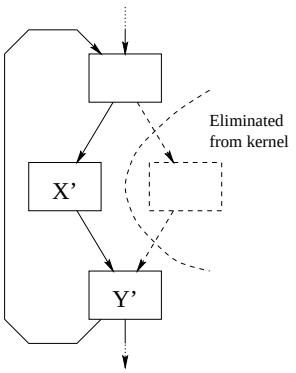


Figure 4.2: Kernel-domination example; X' kernel-dominates Y' even though X did not dominate Y in the original loop.

here is on the loop scope. To distinguish, the ideas of *kernel-domination* and *kernel-post-domination* are introduced.

Basic block X kernel-dominates basic block Y when every path in the kernel from the loop entry block to Y includes X. Similarly, basic block Y kernel-post-dominates basic block X when every path in the kernel from X to any loop backedge includes Y. Note that Y can still post-dominate X even if one or more of the paths from X to Y contain kernel exits. Thus even if Y kernel-post-dominates X, an iteration that takes an exit could execute X but not Y.

Note that excluded paths are ignored in the calculation of kernel-domination and kernel-post-domination. Because of this, there are often cases where even though X did not dominate Y in the original loop, the copy of X does dominate the copy of Y in the hardware kernel since the alternative path has been eliminated (**Figure 4.2**).

Even when no paths are excluded, kernel-post-domination differs from traditional post-domination as shown in **Figure 4.3**.

These relations are ultimately of interest concerning pairs of nodes in the DFG. For ease of description, the definitions are extended to DFG nodes as follows: node *nA* kernel-dominates node *nB* if and only if node *nA*'s owning basic block kernel-dominates node *nB*'s owning basic block, and similarly for kernel-post-domination.

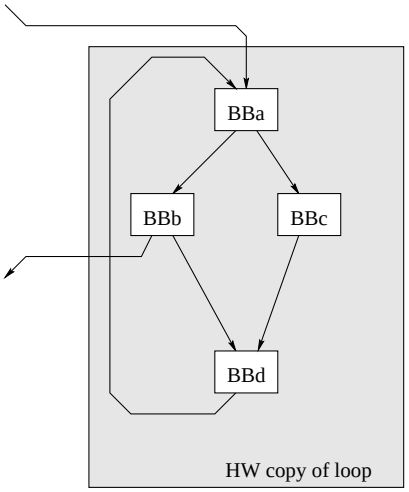


Figure 4.3: Kernel-post-domination differs from post-domination. Basic block BBd kernel-post-dominates BBb but does not post-dominate BBb since BBb has a path to the procedure exit via the loop exit avoiding BBd. In addition, although BBb does not kernel-post-dominate BBd it does post-dominate BBd, since BBd has no path the procedure exit except through BBb.

4.5 Predicates

As the forward build pass progresses, a predicate is recorded for each basic block and each CFG edge. The predicate for a block is recorded immediately before processing the block; the predicate for each outgoing CFG edge from a block is recorded immediately after the block has been processed. The recorded value for each predicate either is a pointer to a DFG node or is null. A null value indicates that the execution of the block or traversal of the edge is unconditional. Otherwise, the output of the indicated DFG node will be TRUE during exactly those iterations during which the basic block would be executed or during which the edge would be traversed. In some cases, the indicated predicate is simply the output of a comparison operator. In other cases a new DFG node is constructed specifically to calculate the predicate, for example, the NOT of a comparison, or an AND of two comparison operations.

The predicate value is treated as any other data value, and the predicate-producing node may be connected via data edges to other nodes as described later. As with any other data depen-

dence, the consumer of the predicate can be scheduled only after the predicate's value has been calculated.

The node to calculate a given predicate is built before it is known if it is actually needed. A subsequent dead node elimination phase trims any predicate calculation nodes that are not used.

Building a basic block predicate (naive version). The predicate for the loop entry block is TRUE. For each other basic block, the predicate is built as the OR of the predicates of all incoming edges (although sometimes a simpler predicate from an already-processed basic block can be reused—see below). When there is just one incoming edge, the calculation degenerates to just copying that edge's predicate.

Building a block predicate (smart version). Block predicates calculated naively are often more complicated than necessary, particularly after different control flow paths merge back together. Traditional logic optimization could be applied to simplify them.

A different approach is used by `garpcc`. It instead looks for an earlier basic block that is executed under exactly the same conditions each iteration, and uses the earlier block's simpler predicate. An earlier block X is known to execute under the same conditions as block Y when both X kernel-dominates Y *and* Y kernel-post-dominates X. The existence of a kernel exit on a path between X and Y does not interfere with this simplification; when the exit is taken, the predicate for Y does not matter.

If no such earlier block executing under the same conditions exists, the block predicate is the "naive" one described above.

Building an edge predicate. For a basic block with two outgoing edges both contained in the hyperblock, the predicate for each outgoing edge is built as its source block's predicate ANDed with the branch condition under with that edge is taken. All other block types have a single outgoing edge in the hyperblock; for these, the edge predicate is the same as the source block's predicate. In the particular case of an Exit block, the approach of *not* ANDing the condition to remain in the hyperblock to create the edge predicate corresponds to "partially resolved predicates" rather than

"fully resolved predicates" [SMJ99]. With partially resolved predicates as used here, predicate calculation is simplified, but some scheduling flexibility is lost since non-speculative (non-safe) operations cannot be reordered with exits.

4.6 Initial values

As the forward pass progresses, it is likely that a use of a variable may occur before it has been defined on any path from the kernel entry to that point. In this case an 'initial value' DFG node is used as the source. Such a node is created for that variable the first time it is needed; subsequent uses before definitions of that variable on that or other CFG paths reuse the same node. The phony definition is considered to have occurred at the very start of the loop entry block. There is a single initial value list data structure relating variables to their initial value DFG nodes.

4.7 Most recent definition of a variable

The concept of the most recent definition of a variable is crucial to the DFG building algorithm. At each 'point' during kernel CFG traversal / basic block processing, there is exactly one most recent definition for any variable. Such a 'point' occurs at the beginning and end of each basic block, as well as between each consecutive pair of instructions within a basic block. This definition of 'point' is not based on C sequence points [HS95].

The DFG building algorithm uses a `lastDefs` data structure, a list that records for each variable which DFG node represents the most recent definition. A `lastDefs` list is always associated with a particular CFG point. The `lastDefs` list is constructed lazily, so that entries are added only as needed—when a variable is used or defined. Auxiliary information can be associated with each entry on the list.

An entry in the `lastDefs` list is created or updated whenever an instruction assigns to a variable. The `lastDefs` for that variable will point to the DFG node resulting from the translation of that instruction. A `lastDefs` entry is also created when a variable is used before it is defined; in that case, a placeholder 'initial value' DFG node is created, to which the new `lastDefs` entry

will point.

At the beginning of processing of the entry block, the `lastDefs` list is empty. For other basic blocks, the initial `lastDefs` list is created before processing of the block begins (Subsection 4.10). In either case the `lastDefs` list is updated during processing of the block; at the end of the basic block, the resulting `lastDefs` list is saved as the 'final' `lastDefs` list for that block. It will be used in creating the initial `lastDefs` list for each of the block's successors.

The concept of 'most recent definition' and associated definition merging algorithms have obvious relationships with reaching definitions [ASU86] as well as static single assignment (SSA) [CFR+91]. In fact things are simplified here because the analysis is always local to the loop and only needs contend with one level of looping, thus only one forward pass is required. There exist marginal similarities with a recent formulation of predicated SSA [CSC+00] although that work is tailored to a fully-predicated representation and has many important dissimilarities.

4.8 Scalar variables in memory (register promotion)

If a scalar variable might be written or read through a pointer access, in general it must "reside" in memory. When this occurs, each use of the variable requires an explicit load from memory, and each definition requires a store. This is detrimental to performance and should be avoided if at all possible. The SUIF representation at this point does not indicate whether a variable (and more specifically, a span of accesses to a variable) must be memory-based. Thus `garpcc` is burdened with performing more accurate analysis to determine whether and when a variable can be migrated to local register storage (register-promoted) versus residing in memory.

The approach used by `garpcc` occurs in two phases. Before DFG construction for a kernel begins, the compiler flags scalar variables that can be register-promoted for the entire kernel. This section describes this first phase. After DFG construction, for variables that were not register promoted for the entire kernel, the compiler performs per-access elimination where possible. These optimizations are described later in Subsection 4.16.

For the kernel under consideration, variables that can be migrated to a register for the

entire execution are flagged. Variables that cannot be register promoted have both of the following characteristics:

- The variable might be accessed (read or written) indirectly through a ‘maybe’ pointer access anywhere in the HW kernel.
- The variable is either definitely or maybe modified during kernel execution.

How does the compiler determine whether a variable “might be accessed indirectly in the kernel”? Pointer analysis [Wil97] is used if possible to see if any of the pointer accesses in the kernel could access the variable. If no pointer analysis is available, a very conservative approach is taken: a variable considered to be “might be indirectly accessed” when both (i) its address is taken somewhere in the program, and (ii) there exists any pointer access within the kernel. With separate compilation, a global variable with external linkage (one that is visible in other files) must always be assumed to have its address taken: even if its address is not taken in the current file, its address might be taken in a procedure in another file and then passed to a procedure in the current file. However, `garpcc` can also be used in a linked compilation mode in which all source files are processed simultaneously. With linked compilation, global variables can be analyzed more precisely: a variable is considered to have had its address taken only when its address actually has been taken in one or more of the files. For cases where pointer analysis cannot be run, this makes a significant difference.

This initial coarse-grain decision of whether for a specific kernel a variable must live in memory is very similar to the *register promotion* described by Cooper and Lo [CL97]. As in Mahlke’s case, maybe-accesses or modifications to a variable occurring on paths excluded from the hyperblock loop do not inhibit register promotion. Because only scalar variables are targeted, it is less general than Mahlke’s *global variable migration* [Mah92], which considers also structure and array elements for register allocation.

4.9 Processing a basic block

The instructions in the basic block are processed sequentially, typically creating one new DFG node for each instruction. After the DFG node has been constructed, data edges are constructed between it and the DFG nodes producing its operands. If the operand is the result of another instruction, pointers are used to track down the corresponding source DFG node, and a data edge is created from the source DFG node to the current DFG node.

If an instruction's operand is a variable that does not need to live in memory (Subsection 4.8), the source DFG node is looked up in the `lastDefs` list. This records the most recent definition *within the kernel* for each variable at that point. There is exactly one such definition for each variable at a given point; multiple prior definitions will have been merged into one as described in Subsection 4.10, and if none exists, an 'initial value' node is created and used.

If an instruction's operand is a variable that needs to live in memory, two additional nodes need to be created: a load, and the node to supply the variable's address to the load. The new Load node is connected as the appropriate input.

If the basic block has a predicate, any non-speculative operations from that basic block must use that predicate as an enabling input. Currently the only types of operations that are non-speculative are stores and exits. A compiler switch can be turned on to inhibit speculative loads, in which case the predicate is also attached to loads. A smarter but unimplemented approach would use a restricted form of predicate promotion [Mah96] for loads, intelligently deciding on a case-by-case basis which loads should be predicated.

If the instruction assigns its result to a variable that does not need to live in memory, the DFG node is recorded as the last definition of that variable in the current `lastDefs` list. A pointer to the DFG node is also attached directly to the original instruction for use when processing direct consumers of that instruction. On the other hand, if the instruction assigns its result to a variable that does need to live in memory, the DFG node's output is used as the data input of a new store node that writes to the variable's memory location.

A copy—assignment from one variable to another—requires no action other than updat-

ing the `lastDefs` list. For an assignment ‘a=b’, the `lastDefs` entry for ‘`a`’ is updated to the `lastDefs` entry for ‘`b`’. But if the assignment involves an explicit or implicit type conversion, it will be treated as any other unary operator: a type conversion node is created with its input from the `lastDefs` entry for ‘`b`’; then the `lastDefs` entry for ‘`a`’ is updated to be the output of the new node.

Exit blocks—those having an outgoing CFG edge exiting the kernel—have their final conditional branch translated to an Exit node in the DFG. The Exit node’s data input is the (possibly inverted) value from the appropriate comparison DFG node. The Exit node also has predicate input—that of its owning basic block—so that it is only enabled during the appropriate iterations. Unconditional branches as well as non-exit conditional branches, no longer serving any function, are simply ignored.

4.10 Merging data values: mux insertion

At a basic block with multiple incoming CFG edges, a given variable may have differing definitions arriving via the edges as indicated by the edges’ respective `lastDefs` lists. When this occurs, a multiplexor (mux) structure is constructed in the DFG to route the appropriate definition for that iteration to subsequent consumers. The data inputs to the mux structure are the distinct data definitions from the arriving `lastDefs` lists; the selector input(s) are derived from predicate logic. The single data output of the mux structure becomes the definition of the variable entered in the `lastDefs` list at the start of processing that block.

Definitions of a variable need be merged—and a mux structure constructed—only when the variable is live at the start of the basic block. It does not matter whether the variable is actually used in that particular basic block.

Finding distinct data inputs Commonly, more than one incoming CFG edge will carry the same last definition for the variable. This would occur for example with two CFG paths that forked and there has been no definition of the variable on either path since the fork. The mux structure needs only a data input for each distinct data source, not each each incoming CFG edge. This assumes

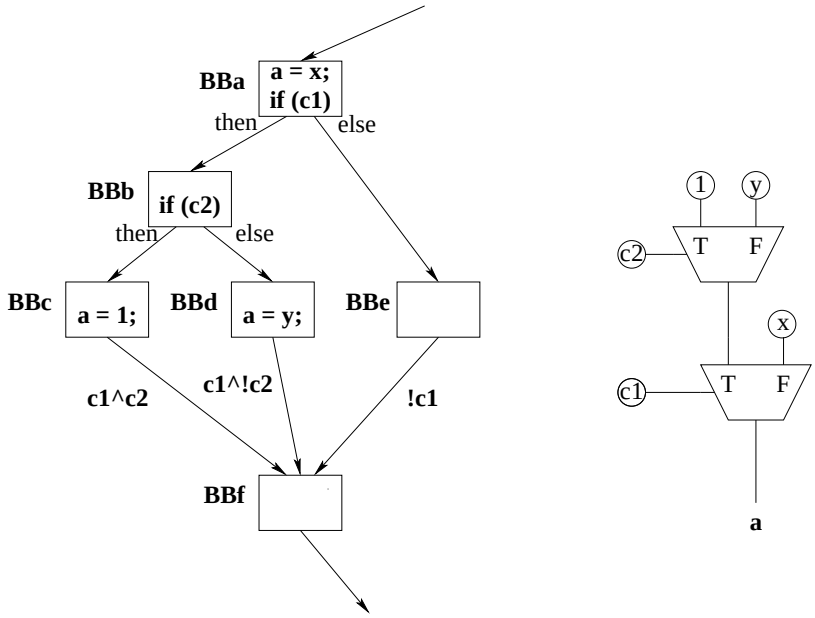


Figure 4.4: Definition merging example.

that a better implementation results from reducing the number of distinct data inputs to the mux structure even at the expense of more complicated control logic.

Two definitions usually count as different if they come from different DFG nodes. The only exception is that two different Constant nodes with the same value count as the same definition; this consideration is necessary only because common subexpression elimination is not applied to constants. If one or more of the incoming `lastDefs` lists have no entry for the variable, an initial value definition is created for that variable if one does not already exist; this node is used as the source for all cases where there was no prior definition.

Decomposition to 2-input muxes The mux structure's function is to route distinct data input d_i to the output when selector input sel_i — the OR of the predicates of the CFG edges along which d_i arrives — is true. At most one of sel_i is true any given iteration; if none is true, the output of the mux does not matter.

The current back-end synthesis flow of Garp currently understands only 2-input muxes,

which have one Boolean control input choosing between the two data inputs. Thus when there are more than two distinct data sources to be merged, a tree of 2-input muxes must be constructed.

The CFG structure is used to guide the topology of the tree of muxes. An example of this case is illustrated in **Figure 4.4**. The general principle here is to have the constructed mux tree reflect, in reverse, the forking in the CFG. It is desired to first merge a pair of values that arise from paths that forked closest to the current merge point. Of the three different definitions reaching BBf, the ones from BBc and BBd are chosen to merge first. This decision is made on the fact that their least common dominator BBb is closer to the merge point than the least common dominator of any other pair (BBa in both cases). Then the result of that merge is merged with the remaining data input.

The CFG structure is also used to simplify the control inputs at each mux. A key idea is this: when a mux is being constructed, if the CFG edges conveying its two data inputs are all dominated by a basic block block D other than the kernel entry block, then the block predicate for D can be eliminated from the expression controlling the mux. The reasoning is this: when $blockpred(D)$ is false, then no paths going through the mux matter, and the control input can be anything and still be correct. Thus the mux only matters when $blockpred(D)$ is true, so the control logic may as well be simplified to assume $blockpred(D) = 1$. In the example, since BBb dominates both source edges $BBc \rightarrow BBf$ and $BBd \rightarrow BBf$, the block predicate for BBb, $c1$, can be factored out of the select input for the first mux, leaving just $c2$. This is in effect performing "don't care" optimization on the Boolean expression controlling the mux [DGK94].

OR simplification Even if the above case does not apply, there are other ways the OR of the incoming edges can be simplified by scanning the DFG near the nodes producing the incoming edge predicates. The following patterns are recognized and simplified:

(p AND q) OR (p AND NOT q) → p
(p) OR (NOT p AND q) → p OR q

When there is a list of more than two values to OR together, the list is scanned for any pair matching either case above; if one exists, it is reduced. Only when neither occurs is straightforward OR construction used.

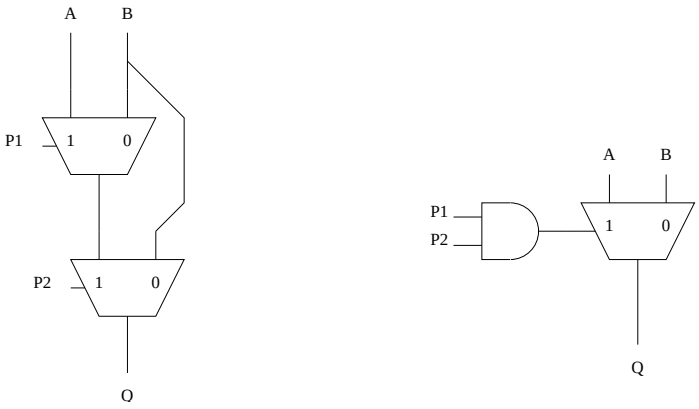


Figure 4.5: Example of mux collapsing.

Mux collapsing The above approach recognizes when multiple CFG edges carry the same definition and simplifies mux structure appropriately, but its scope is limited to a single merge point. More work is performed later, after DFG construction is complete, to recognize similar situations that resulted from two different merges.

The DFG is examined for pairs of successive muxes such that the group has only two distinct data inputs coming from outside (**Figure 4.5**). In this case one of the muxes can always be eliminated at the cost of slightly more complex control circuitry.

This optimization still has a small scope and can easily be inhibited when one of the original merges required the building of a mux tree, in which case the shared data input might not feed adjacent muxes. In retrospect, immediate decomposition of merges into 2-input muxes is not the best approach. Better would to be to retain the fully-decoded mux representation [BG02], which have any number N data inputs $d_{0 \leq i < N}$ and N control inputs $c_{0 \leq i < N}$; input d_i is chosen when c_i is true; no more than one of the control inputs may be true any iteration. This representation allows easy collapsing and simplification of successive muxes. Eventual decomposition to 2-input muxes should occur later in the flow, or even in the target-dependent synthesis phase.

Other mux simplification After DFG construction is complete, other optimizations are performed to eliminate muxes when the two data inputs are identical or when the control input is

constant; these situations arise as a result of other optimizations. There is a special set of optimizations for the muxes that result from SUIF's somewhat convoluted representation of the C language's short-circuiting semantics for logical expressions; these are described in Subsection 4.15.

4.11 Building precedence edges

To aid with the construction of precedence edges, HWSSA maintains lists of all "upstream" store, load, and exit nodes—those built during processing along any CFG path from the start of the kernel entry block to that point. As with the `lastDefs` list, these lists are always associated with a particular point in the CFG. The memory access lists include both those accesses originating from explicit memory accesses in the original program as well as accesses introduced for scalar variables that must live in memory. Before processing a basic block that has multiple incoming edges, each list is calculated as the union of the respective final lists of the predecessor basic blocks.

All potential sources of precedence edges within the iteration are located in the lists of upstream loads, stores, and exits. These lists contain only nodes that occur on CFG paths that could reach the current point from the top of the kernel. This helps eliminate unnecessary precedence edges; for example, a precedence edge should not and will not be constructed from a store in an IF statement's THEN branch to a load in the same IF statement's ELSE branch. The IMPACT compiler [Mah96] also exploits cases where two operations cannot execute the same iteration, but it analyzes the involved predicates directly rather than using the original CFG and communicating that information to the DFG structure.

A DFG node's incoming precedence edges are constructed based on its opcode:

A DFG node with opcode:	...needs precedence edges from:
load	upstream stores
store	upstream loads, stores, and exits
exit	upstream stores and exits

If the node is a load, store, or exit, it is then added to the appropriate list.

This step is grossly conservative, for example adding a precedence edge from store to each subsequent load even if the load is from a different array. Later phases utilize dependence

information to remove precedence edges which are false.

In cases where correct ordering is otherwise guaranteed—typically by a path of one or more data edges between the two nodes—the precedence edge is redundant and may later be deleted for efficiency reasons (Subsection 4.17). This must be the very last step, after all other optimizations have been performed. Otherwise necessary ordering information could be lost.

4.12 Forming loop-carried data edges

After each basic block has been visited on the forward pass, the next task is to create data edges carrying the final values of variables at the end of an iteration back to the beginning of the (next) iteration.

In general there can be multiple CFG backedges. When a variable has different most recent definitions along different backedges, they must be merged. The process of merging values from different backedges is exactly the same as for merging multiple edges incoming to a basic block (Subsection 4.10), and is analogous to 'backedge coalescing' in the IMPACT compiler [Mah96].

After this merging there is a single final definition for each variable—a 'final' `lastDefs` list. Each entry in the final `lastDefs` list is then matched with the list of initial values. When a variable is found in both lists, the `initVal` node is converted to a Hold node, and a data edge is added connecting the final source node to the input of the corresponding Hold node.

A variable's presence in the initial value list but not in the final `lastDefs` list indicates that the variable does not change value during kernel execution (is kernel-invariant). A variable could be kernel-invariant even if it is not loop-invariant in the original natural loop; this case occurs when all modifications of the variable occur on paths excluded from the kernel. In the kernel-invariant case, the `initVal` node is converted to an Input node, and no loop-carried data edge is constructed. The Input node will be initialized once, before the start of kernel execution.

If a variable occurs in the final `lastDefs` list but not in the `initVal` list and is not live on any backedge, that means that the variable is never used before it is defined within an iteration. Thus no loop-carried edge need be constructed in this case either.

```
do {
    b = a + b;
    if (b > 10) break;
    c = b + 1;
} while (1);
<uses of a, b, and c>
```

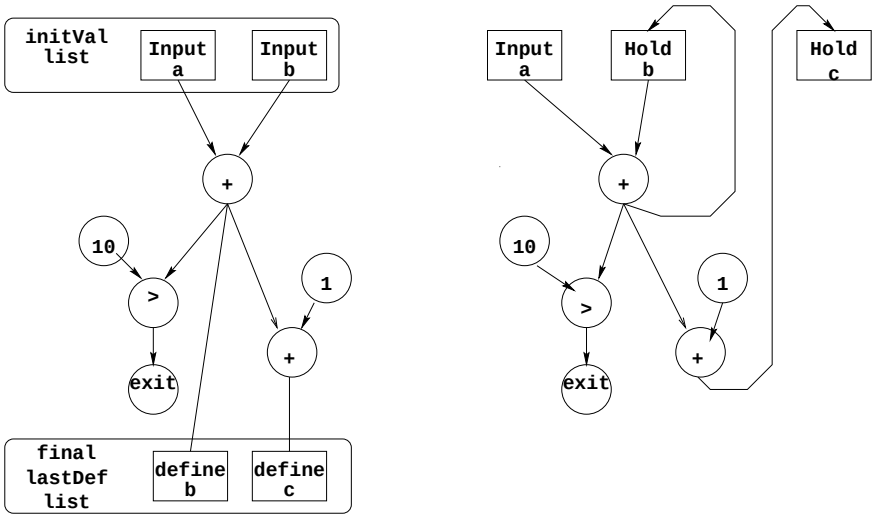


Figure 4.6: Hold nodes introduced for regular loop-carried variable ‘b’ and for loop-carried live value ‘c’. Middle graph shows state at end of forward pass. Right graph shows state after formation of loop-carried data edges.

A rare case exists for variables that are live on a back edge, but don’t occur in the `initVal` list. This situation occurs for variables that are not used before being defined within the loop, but are live at some exit that occurs before a definition. In other words, a definition in one iteration is the one that is live at an exit during the next iteration. In these cases, a Hold node must be added for the variable. The Hold node provides the correct version of the variable to copied out of the array at kernel exit.

An example illustrating all cases is shown in **Figure 4.6**. Variable ‘a’ is used but not defined, and so becomes a kernel-invariant input. Variable ‘b’ is both used and defined, and so gets its own Hold node. Variable ‘c’ is not apparently used, but because it is live on the loop backedge, the loop carried value must be live at an exit; thus it also gets a Hold node. In this case a liveness edge will eventually be built from the Hold node to the exit at which it is live.

4.13 Forming loop-carried precedence edges

A loop-carried (distance $d > 0$) precedence edge from node *nA* to node *nB* indicates that the execution of *nA* in iteration i must occur before the execution of *nB* in iteration $i + d$. The value of d is usually 1.

Such edges are needed for scheduling only when pipelined scheduling will be performed. They are not needed with non-pipelined scheduling since in that case, every operation in iteration i executes before any operation in iteration $i + 1$ does. However, these edges are useful in other transformations, namely redundant load elimination and queue use recognition (Chapter 5). Thus loop-carried precedence edges are calculated in all cases.

Recall that within an iteration, precedence edges are not required between a pair of nodes that cannot occur on a single control path through the CFG. No such limitation exists for loop-carried precedence edges, since in general any control path in one iteration can be followed by any other control path in the subsequent iteration. Therefore, a loop-carried precedence edge is constructed between *every* pair indicated in the following table:

A DFG node with opcode:	...needs a loop-carried precedence edge from :
load	every store
store	every load, store, and exit
exit	every store and exit

However, if an intra-iteration (distance 0) edge already exists between the two nodes, the less constraining loop-carried edge need not be formed.

Thus the calculation of loop-carried precedence edges is trivial, but leads to potentially many edges. As with intra-loop precedence edges, a subsequent step will use dependence analysis to remove false loop-carried precedence edges when possible. Furthermore, redundant loop-carried precedence edges will be removed as the final step.

4.14 Live variables at exits

This phase determines for each exit which values must be copied back to the main processor when that exit is taken. A liveness edge is constructed for each such variable, from the node responsible for the reaching definition to the exit DFG node. Only variables that have been register-promoted need to be copied out of the array.

The following tests determine which variables must be copied back. The variables that must be copied back are those that satisfy both of the following conditions:

1. The variable is live on that CFG edge exiting the kernel.

2. The variable is modified anywhere in the kernel.

If (1) is true but not (2), then the software version of the variable is still up to date, and no copy needs be performed. If (2) is true but not (1), then the variable is not used in subsequent computation before it is defined again, and no copy needs be performed.

Figure 4.7 illustrates different cases and how they are handled.

If the variable is modified (or used) on any path from the loop entry to the exit under consideration, then there will be an entry for that variable in the `lastDefs` list at the end of the Exit block. The indicated DFG node is the one providing the value for the variable, so the edge is constructed from that DFG node to the exit DFG node. See '`a`' in **Figure 4.7**.

On the other hand, when no definition of the variable occurs between the loop entry and the exit, then the definition live at the exit is the loop-carried version of the final definition that reaches the end of the loop. In this case a liveness edge is added from the variable's Hold node to the Exit node. See '`c`' in **Figure 4.7**.

An Input node is never be the source of a live variable output for the same variable. Since such a variable must be kernel-invariant, value existing prior to kernel execution is still valid. See '`b`' in **Figure 4.7**.

In contrast, an Input node for variable '`v1`' can be the source of a live variable output for a different variable '`v2`'; this situation results from a copy (variable-to-variable assignment). A constant node can also be the source of a live variable output for a variable '`v2`'. In either of

```
do {
    a = a + b;
    if (a > 10) break;
    c = a + 1;
} while (1);

<uses of a, b, and c>
```

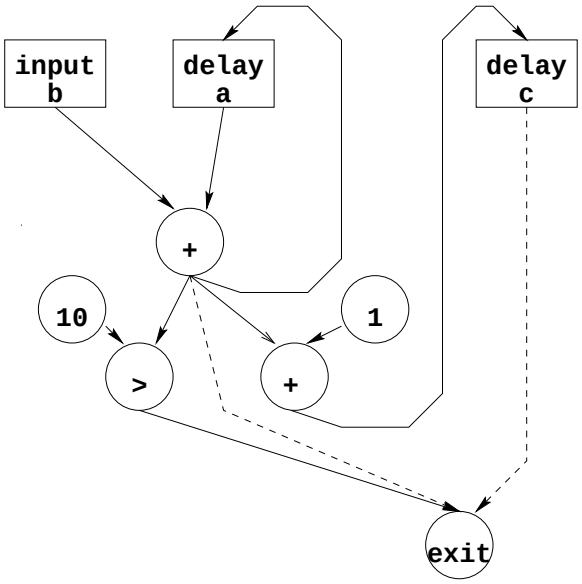


Figure 4.7: Addition of liveness edges.

these two cases, the liveness edge is still added. However, the final patching algorithm is smart enough to realize that no actual data needs be transferred from the array in either case. At the appropriate software reentry point, `garpcc` simply inserts the appropriate assignment, for example "`v2 = v1;`" or "`v2 = 2;`".

4.15 Miscellaneous DFG optimizations

Many optimizations are performed both during DFG construction and as separate passes over the completed DFG. For example, some common subexpression elimination (CSE) is performed during the forward pass of DFG construction ([ASU86] Example 5.9), but another CSE pass is performed on the entire DFG later after other optimizations have been applied.

Fortunately, the implementation of most optimizations on the DFG representation is straightforward since control flow has been eliminated. Thus, no global dataflow analysis framework [ASU86] is required; the local (within basic block) version can be directly applied to the entire kernel. Performing optimizations on the DFG representation can also be compared to per-

forming them on SSA form, because (i) all data definition/use relationships are explicit, and (ii) all variable/register-based write-after-read (WAR) and write-after-write (WAW) hazards have been eliminated.

Although many of the same optimizations were performed earlier on the original software representation of the program, additional opportunities arise here after paths have been excluded from the kernel. Variable definitions and memory accesses on those paths that had prevented the optimizations now essentially disappear from consideration, giving new opportunities. Additional opportunities arise from the fact that most operations are promoted to unconditional execution.

Common subexpression elimination Common subexpression elimination (CSE) is a well-known optimization for identifying and removing redundant computation: identical operations on identical operands. When a node has "identical operands" it is immediately obvious from the structure of the graph. All simple operator nodes are subject to elimination, as are all nodes introduced to support predicated execution (Boolean calculations and muxes). Store and Exit node types are not considered for elimination. A Load can be eliminated using another Load if there is no intervening store (the exact definition of "intervening store" is given in Subsection 4.16). When a load is eliminated, its impinging precedence edges must be added to the equivalent remaining load.

A standard SUIF pass (`porky -cse`) performs simple CSE early in the compilation flow. However, it sometimes misses cases where the identical operations occur in different basic blocks, particularly when they are on alternative control paths. This is not a fault since with software execution only one or the other operation will actually be executed. However with the fully speculative execution used on hardware, both would be implemented in hardware and performed every iteration, so eliminating one is clearly beneficial. Because the fully-speculative approach allows CSE to combine operations on alternative control paths in the original loop, the result is similar to Mahlke's *instruction merging* [Mah96], except that the case here does not handle non-speculative operations—stores and exits.

The synthesis backend for Garp may in fact reverse some of the work that CSE has done. But the decision to undo sharing—to replicate subtrees in the DFG to help technology mapping—is

a target-dependent decision and so is appropriate for the back end.

Kernel-invariant expression elimination A kernel-invariant node is one that is guaranteed to compute the same result each iteration of a particular kernel execution due to the fact that its inputs are also the same each iteration. In such cases, the operation can be moved outside the kernel immediately before kernel entry (in software), and thus computed only once per kernel entry rather than once per iteration. During the preliminary software phase of compilation (Chapter 2) a loop-invariant optimization pass was performed (`porky -loop-invariants`). Also, a special case optimization based on loop-invariant conditionals has been performed (`porky -loop-cond`). Additional opportunities are available at this point because variable definitions on excluded paths no longer interfere with this optimization.

The first phase of the optimization is to mark DFG nodes that are invariant. Nodes are visited in forward topological order. Input nodes are by definition kernel-invariant; they represent variable values that are used but not modified within the kernel. Obviously, Constant nodes are also kernel invariant. Thereafter, a node is marked as kernel-invariant if all data inputs are kernel-invariant. All simple operator nodes are potentially kernel-invariant. Comparisons, predicate calculations, and muxes are also subject to be marked as loop-invariant. Store and Exit nodes are never marked as kernel-invariant. A Load node is marked as kernel-invariant if, in addition to its address input being invariant, it has no precedence edges to or from a Store node.

The second phase creates a new Input node for each invariant node that has a non-invariant consumer. The invariant expression for each created Input node is recorded on that node; because it is executed in software, it is expressed as C code, and thus the DFG subtree producing the value needs to be converted back to a C expression to be evaluated in software immediately before kernel entry. The consumers of the invariant node are then modified to instead use the newly created Input node. Dead node elimination later eliminates the unneeded invariant DFG nodes.

This optimization usually reduces area and can also reduce the critical path, which is primarily of importance with non-pipelined execution. However, it also has the potential to increase the overhead of kernel use. In fact, the latency of the one-time computation of the invariant expression

in software may be greater than the total time savings to calculate the expression every iteration in hardware, particularly when the iteration count is low and/or the invariant computation is not on the critical path. The current implementation, however, assumes this optimization is always beneficial and applies it whenever possible.

Using conditional branch information Information from the comparisons controlling conditional branches can be exploited during DFG construction. For example, in the fragment

```
if (x == 0) {
    y += x;
} else {
    y -= x;
}
```

the value of x in the THEN branch is necessarily zero, and the addition can be eliminated. In general such cases can enable further constant folding, identity simplification, and mux simplification. Humans would rarely write such code since they would usually perform the equivalent optimization themselves. However, new cases can arise due to path exclusion.

The algorithm works as follows. The `lastDefs` list at the begining of the THEN branch is modified: the `lastDefs` entry for '`x`', in addition to indicating the source DFG node, is also annotated to indicate that the value is known to be 0 at that point. When such an annotation is present, that constant can be substituted for any use of the variable, as when supplying the operand to the addition. However, at merges it is best to use the original node. Again refer to the above code snippet. Consider the merge point at the end of the IF statement. If the `lastDefs` entry for '`x`' on the THEN branch were simply overwritten with a new constant 0 node, then at the merge point the two reaching definitions of '`x`' would be different, and a mux would be inserted. Keeping the original node allows the merge algorithm to recognize that it is in fact the same definition from both branches. However, since one of the definitions has no "known constant" annotation, the entry in the merged `lastDefs` list also has no such annotation. The "known constant" annotation is propagated to the entry in the merged `lastDefs` list only if all reaching definition entries have a "known constant" annotation, and all the values are the same.

This optimization was not used frequently with the studied benchmarks. It was never used in SPECint95, and used just once in the wavelet image compression benchmark.

Constant folding Constant folding is simply the reduction of expressions of compile time constants to the equivalent constant. Again, opportunities increase when definitions on excluded paths need not be considered. With software compilation constant folding usually requires constant propagation to achieve its full potential; however here an explicit constant propagation pass is not required as it is performed implicitly in construction of the DFG representation.

It could be argued that much of the benefit of constant folding is also realized by invariant expression elimination, since any constant expression is necessarily loop-invariant. In both cases the same active operations can be removed from the DFG. However, with constant folding, the expression is replaced with a compile-time constant, rather than a loop-invariant input. A compile time constant usually allows better synthesis. For example, consider the case of implementing a shift on the Garp array. If the shift amount is a compile time constant, the shift can be implemented using hard-wired routing in just one row, or even with no additional rows if the shifted data source and consumer are adjacent. In contrast, if the shift amount is only kernel invariant, the variable shift module must be used, which can consume up to three rows.

Mux elimination Application of CSE or constant folding may lead to a situation where the two data inputs to a mux are in fact equivalent (the same node or the same constant). In such cases the mux can be totally eliminated; the mux's data consumers instead use the mux's common data input.

Similarly, if constant folding or Boolean simplification reveals that the mux's control input is a compile-time constant, the mux can be eliminated, with consumers instead being connected to the appropriate data input.

Identity simplification Integer operations that add or subtract zero, shift by zero, or multiply by one are eliminated. Similar optimizations on Boolean values are described later.

Boolean value identification The C language defines signed and unsigned integer data types of various sizes. The `garpcc` compiler faithfully retains such type information so that necessary truncations and sign extensions are performed on data.

ISO C [HS95] does not contain a Boolean data type. Although the result of a comparison is defined to be either 0 or 1, the type of the result is signed integer, which is implemented as 32 bits on the Garp MIPS core. However, no information is lost if only a single bit is used to carry the result. This can be exploited to advantage in hardware. Therefore `garpcc` identifies as 'Boolean' those operations guaranteed to produce only 0 or 1. When necessary, Boolean values are correctly converted back to a standard C type.

The algorithm identifies "base case" Boolean nodes: comparisons, constant '0', and constant '1'. Then it propagates the Boolean property to nodes that have an appropriate opcode and have all inputs already flagged as Boolean. Appropriate opcodes include bitwise AND, OR, XOR, as well as muxes. Opcodes that are not appropriate include bitwise NOT and addition. However, all predicate calculations are marked as 'Boolean' when they are constructed, including NOT operators.

The synthesis step targeting Garp (Chapter 7) heavily exploits Boolean values. Such values are computed and routed in a separate column of the array, thus not consuming resources in the main datapath. Furthermore, operators marked as Boolean are subject to special optimizations as described next.

Optimizations on muxes resulting from Boolean expressions The C language's short circuiting semantics regarding two-input logical operators ("&&" and "||") dictate that their right operand is not evaluated "if the value of the first operand provides sufficient information to determine the value of the expression" [HS95]. Expressions with these operators result in certain SUIF idioms that in turn result in Boolean muxes in the DFG. An example is shown in **Figure 4.8**.

The first IF/THEN/ELSE statement in **Figure 4.8** results in a mux producing the current value of '`suif_tmp2`'. The Boolean mux generated in such cases has the characteristic that its control input ('`suif_tmp1`' in this case) comes from the same source as one of its data inputs (the control input might be inverted; inversions are eliminated by swapping the mux data inputs). An

Original C code:

```
if (i<100 && *p) {
    sum += *p;
}
```

SUIF representation, translated back to C:

```
suif_tmp1 = i < 100;
if (!suif_tmp1) {
    suif_tmp2 = suif_tmp1;
} else {
    suif_tmp2 = (*p != 0);
}
if (suif_tmp2) {
    sum = sum + *p;
}
```

Figure 4.8: Short-circuiting semantics in SUIF.

optimization pass recognizes such muxes and replaces them with the equivalent 2-input Boolean function, as shown below.

mux inputs (all Boolean)			equivalent expression
c	d0	d1	
x	y	x	x OR y
not x	x	y	x OR y
x	x	y	x AND y
not x	y	x	x AND y

A related optimization recognizes and eliminates comparisons of a Boolean value with 0, as below.

Original Expression (x known Boolean)	Simplified Expression
$x == 0$	NOT x
$x! = 0$	x

Application of these optimizations results in a straightforward data calculation—much as a human designer would produce—despite the rather complicated SUIF intermediate step. Note that

`garpcc`'s speculative execution model in some sense undoes the short circuiting semantics of the expression: the second comparison is performed regardless of the result of the first comparison. Thus both can be performed in parallel, reducing the critical path. If the second part of a Boolean expression includes a non-speculative operation, unlike in the example shown, that operation would have the appropriate predicate from the first part of the expression attached.

Other Boolean optimizations In some unexpected cases, combinations of control flow and variable assignments in the SUIF representation led to generated predicate logic where an AND or OR gate had a constant 0 or 1 input. Such cases are recognized in the DFG and transformed to eliminate the gate. Either the non-constant input or the constant 0 or 1 is substituted to feed to original consumers of the gate's output. The elimination of the gate may lead to additional elimination of DFG nodes both backwards (through dead node elimination) and forward (through reapplication of this optimization, mux elimination, and/or constant folding).

Negation pushing and elimination A negation operator may cost an entire row in a Garp configuration. Therefore it is useful to eliminate negations when possible by combining them with additions or subtractions, or by "pushing" them towards outputs until they are adjacent to another negation and cancel each other out.

The following peephole transformations are performed repeatedly on the DFG until no more changes occur. Fanout from intermediate nodes prevents the application of these transformations.

Transformed from	To
neg(neg(a))	a
add(a, neg(b))	sub(a, b)
add(neg(a), b)	sub(b, a)
sub(a, neg(b))	add(a, b)
sub(neg(a), b)	neg(add(a, b))
neg(sub(a,b))	sub(b, a)

In retrospect, a smarter approach would collapse a tree of additions, subtractions, and negations to a single summation node, keeping track of which inputs are negated. Then this node could be decomposed into a tree of two-input subtractions and additions, with the additional benefit

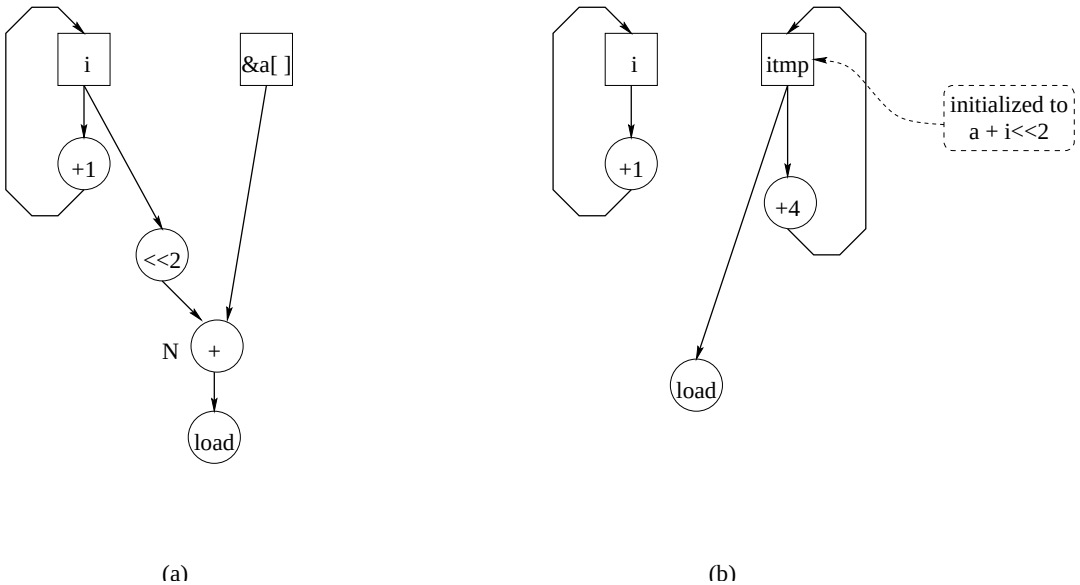


Figure 4.9: Induction variable introduction.

that inputs on a critical path/cycle could be favored, effectively performing a reassociation optimization.

Induction variable introduction A DFG node N whose value changes by a constant amount each iteration is a candidate for replacement by an induction variable. The costs and benefits are easiest illustrated by example.

Figure 4.9(a) shows the DFG before the transformation. The variable `i` is the original loop index, and is an example of a base induction variable, since it is directly incremented by a constant amount each iteration. The value at node N is calculated from `i` in a way that it also changes by a constant amount each iteration. **Figure 4.9**(b) shows how the value at N can be equivalently provided by a new induction variable `tmp1`, which is directly incremented each iteration rather than being calculated from `i`. `tmp1` is initialized with the value that node N would have produced the first iteration—in this case, `&a[0] + i<<2`.

This transform may be beneficial if it reduces the area of final configuration; this situation is more likely to occur when more work is involved in computing N from '`i`'. It can be somewhat

difficult to estimate the area savings even in simple cases. In particular, a shift can be free when it occurs between two adjacent rows, but can cost a row otherwise, so there may or may not be a savings from eliminating a shift.

The transform typically allows N to be calculated earlier in the final schedule, which may be significant for non-pipelined execution.

A more subtle benefit realized during synthesis (and one that is most difficult to quantify) is the easing of routing pressure. The value ‘`i`’ no longer needs to be routed to near the use of N, since N is independently calculated.

This transform introduces a new variable that needs to be initialized. This adds a small amount to the startup cost for using the kernel which must be considered as well. However, in many cases the initialization of a loop-invariant address can be eliminated (in this case, the address of the array `a[]`), so there is no difference.

This transform could be also applied when the increment is kernel-invariant but not constant. However the increment analysis was borrowed from the analysis to look for memory queue uses (Chapter 5). Since queues can only be used with constant stride, that is all the analysis looked for; the analysis has not yet been extended to also recognize kernel-invariant increments.

After initial implementation of this transform, it was found that it hurt as often as it helped. The main cause was a mismatch with the backend synthesis technology mapping. Even if this were corrected, a smart heuristic would still be required to apply the transform only in select cases. Since few cases were seen where great benefit was derived, this transform was simply disabled.

Operator size reduction ISO C semantics [HS95] dictate that arithmetic and logical operations involving type `char` and/or `short` operands be performed at the precision of the `int` type. For example, the C code

```
short a, b, c;
...
a = b + c;
```

results in the following implicit conversions:

```
short a, b, c;
...
a = (short)((int)b + (int)c);
```

During initial DFG construction all three casts are faithfully translated to DFG nodes. But since the destination's representation size of `short`(16 bits) is less than that of `int`(32 bits), the upper bits of the addition are discarded. Thus a 16 bit addition will give the same result as a 32 bit addition in all cases. This in turn means that the addition utilizes just the lower 16 bits of each operand. Thus reducing the size of one operator may enable the size reduction of other operators. Also, there may be type conversions on the operands that can be eliminated, as in this example.

In fact when `garpcc`'s synthesis backend targets Garp, no direct benefit comes from shrinking the addition itself from 32 bits to 16 bits. There is no area savings since the datapath generator never attempts to pack two arithmetic modules on the same row. There is no timing advantage since the Garp array timing model specifies the same cost for any use of the carry chain, whether the span utilized is 4 bits or 40 bits. The actual savings in this example are derived by eliminating the two sign-extending type conversions on '`b`' and '`c`'.

Automated data width analysis [Har77, RS94, BGWS00] could be used to find additional applications of this optimization even when the variables were declared by the programmer to be 32 bits. This has not yet been integrated into `garpcc`.

Dead node elimination A cleanup pass eliminates those nodes that are "dead". A node is dead if it is not "live"—required for proper execution. A node is live if it has side effects, i.e. is a store or an exit. A node is also live if its data output is used by another "live" node, including the case where the node supplies a live value to an exit node.

The algorithm starts by marking as live nodes with side effects: stores and exits. Then it marks as live any node whose output is used by one or more marked live nodes, and so on. Only data and liveness edges need be traversed. Once no more nodes can be marked as live, any remaining nodes not marked as live are known to be dead and can be safely removed.

Note that because of the possibility of cycles of dead nodes, the reverse approach—iteratively marking nodes as dead—would not work. Considering a cycle of nodes X and Y, node X cannot be marked dead until Y is marked dead, but Y cannot be marked dead until X is marked dead. Thus neither would be marked as dead even when in fact they both are.

Operation migration (not implemented) Consider two nodes X and Y, where Y is the only consumer of X. If a prune removes Y, X has no actual consumers left in the DFG. Yet it cannot simply be eliminated; its result will be needed by the software version of Y, and so a liveness edge to the new exit will result. However, the result is used only those iterations during which that particular exit is taken. During all other iterations, the computation is discarded.

Mahlke's "operation migration" optimization recognized this situation and moved operations such as X in this example to the basic block outside of the superblock/hyperblock following the exit. After this transformation, the operation is executed only during those iterations when its result is needed.

The similar optimization has not been implemented in `garpcc` because (i) the situation was was not expected to occur frequently, (ii) optimizations transforming both the DFG and software outside the kernel are more complicated to implement, and (iii) it would not always be beneficial; the migration of a binary operation may increase the number of live values to be copied out at the exit by one, increasing overhead.

However, a preliminary study showed that this situation did in fact arise frequently. At least 54 operations from all kernels could be moved to software using this optimization. The number is conservative since it only counted operations having no consumers in the current DFG; once some operators are moved out of the DFG, others may in turn become eligible.

It is likely that many of these operations were "partially dead" even in the original loop,

and that application of partial dead code elimination [KRS94] would greatly reduce the need to implement the optimization at this point.

4.16 Memory access optimizations

Some simple optimizations involving memory loads were performed in the earlier common subexpression elimination optimization. This section describes more complicated optimizations aimed at removing or coscheduling memory accesses where possible. The steps are presented in the correct order relative to each other, but in the compiler flow they are interleaved with other phases of DFG construction and optimization.

Exclusive stores Stores cannot be speculatively executed—they execute only each iteration that they would have been executed in software. If store S1 and store S2 occur on alternative control paths (are *exclusive* stores), they can never execute the same iteration, even in hardware. Synthesis for Garp can exploit this for better scheduling. In particular, it can schedule both S1 and S2 for the same cycle, which it could not do if they could possibly both execute.

Therefore information regarding exclusive stores is recorded and passed to synthesis. This information is easily computed by analyzing the initial construction of precedence edges. Any two stores that could be executed the same iteration will have a distance-0 precedence edge between them at this stage; therefore, any two stores that do *not* have a distance-0 precedence edge between them must be exclusive, and are recorded as such.

This concept could be expanded to include loads, but only when they are executed non-speculatively, that is, they are controlled by the appropriate predicate. But reverting to non-speculative loads could lengthen the critical path/cycle. A heuristic to decide which loads should be speculative and which should be non-speculative to allow coscheduling has not been developed.

Removal of false precedence edges At this point false precedence edges are removed to the degree allowed by the array and pointer analysis at the compiler's disposal. Each precedence edge constructed during DFG construction is analyzed, and if the analysis guarantees that those two

accesses cannot possibly access the same location during relevant relative iterations, the edge is removed.

Specifically, for two memory accesses connected by a distance-0 edge, if array analysis indicates they will never access the same location during the same iteration, the edge is given a less-stringent distance of 1. In later analysis, a distance-1 edge from access X to access Y will be removed if array analysis indicates that any location accessed by X in any iteration i is never accessed by Y in any iteration $j > i$. Pointer analysis simply says that access X in any iteration can never access the same location as access Y in any iteration and thus can eliminate both distance-0 and distance-1 edges.

The analyses used here, like any practical analyses, are inexact in that they may fail to recognize some false edges, but conservative in that they will never remove a true edge.

Removal of redundant edges (Subsection 4.17) does not occur until after all optimizations on the DFG have been performed.

Definitely same locations For some of the following optimizations, it is useful to know when two memory accesses are *definitely* to the same location, whether in the same or different iterations. The Garp compiler recognizes the following situations:

- Two accesses both address the same scalar variable. This situation results when it was determined that the scalar variable must live in memory as described in Subsection 4.8.

- Two accesses have address inputs from the same DFG node. This situation typically occurs when the two accesses dereference the same pointer and the pointer is not modified between the accesses. Similarly, arrays passed as procedure parameters lead to pointer arithmetic, and after common subexpression elimination, equivalent array accesses are almost always share the same DFG node for the address.

- Array dependence analysis has determined that the accesses are definitely to the same location, either within each iteration, or separated by a fixed number of iterations. That is, access X in iteration i is always to the same location as access Y in iteration $i - C$, where C is a

constant.

Intervening operations Some optimizations described below need to discern whether a certain type of operation necessarily *intervenes* between two other operations opA and opB. The specific operation type of concern depends on the optimization. Fortunately in all cases of concern, if there is an intervening operation, there will be precedence edges from opA to the intervening operation to opB. Therefore a simple check of the graph topology gives an accurate answer.

Furthermore, the precedence edges are present only in cases that are important—they exist only between pairs of memory operations that could occur on the same control path, and only in cases where compiler analysis says they definitely or might access the same location. Operations on paths excluded from the hyperblock are not even part of the DFG and therefore cannot register as an intervening operation.

The algorithm for finding intervening operations is shown in **Figure 4.10**. It is called only for distances d of 0 or 1.

Optimization: removing unnecessary loads A load can be removed when it can be determined that the value that would be loaded is the same as that produced by another DFG node that iteration. The compiler looks for two cases.

The first case is illustrated in **Figure 4.11**. Load L2 can be eliminated; its consumers simply reuse the value from another load L1 because all of the following conditions are true: (i) L1 and L2 definitely access the same location each iteration, (ii) there is no intervening store that might access that location, and (iii) L1 executes every iteration that L2's value is needed. Condition (iii) is always true if loads are always performed speculatively; otherwise, it requires that L1 dominates L2. In general, when the transformation is applied, precedence edges impacting both loads are unioned for the remaining load. This case overlaps with common subexpression elimination in that many situations would be recognized by both optimizations. Also, this case parallels Mahlke's *global redundant load elimination* [Mah92]. As with Mahlke's formulation, a potentially intervening store on an excluded path does not inhibit this optimization.

In the second case (**Figure 4.12**), a load L2 can be eliminated and simply reuse the value

```
// — is there a node nx of opcode optype such that
// — there are precedence edges n1→nx and
// — nx→n2, with distances summing to d?

function interveningOp(optype,n1,n2,d) {
    foreach nx in n1.outgoing_precedence_edges {
        //— nx is the potentially intervening operation
        if (nx.type != optype) continue;
        foreach ny in nx.outgoing_precedence_edges {
            if (ny == n2 and
                dist(n1 → nx) + dist(nx → ny) ≤ d) {
                return TRUE;
            }
        }
    }
    return FALSE;
}
```

Figure 4.10: Algorithm to determine presence of a necessarily intervening operation.

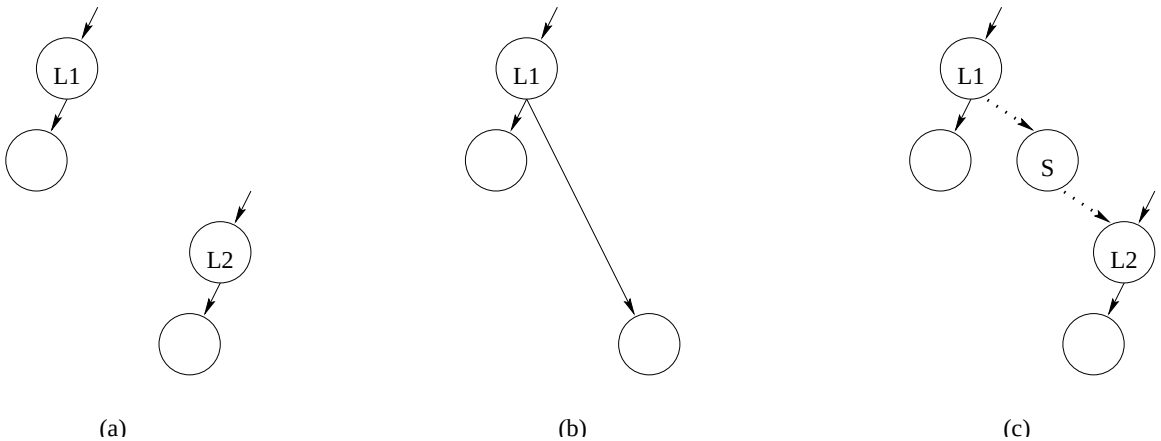
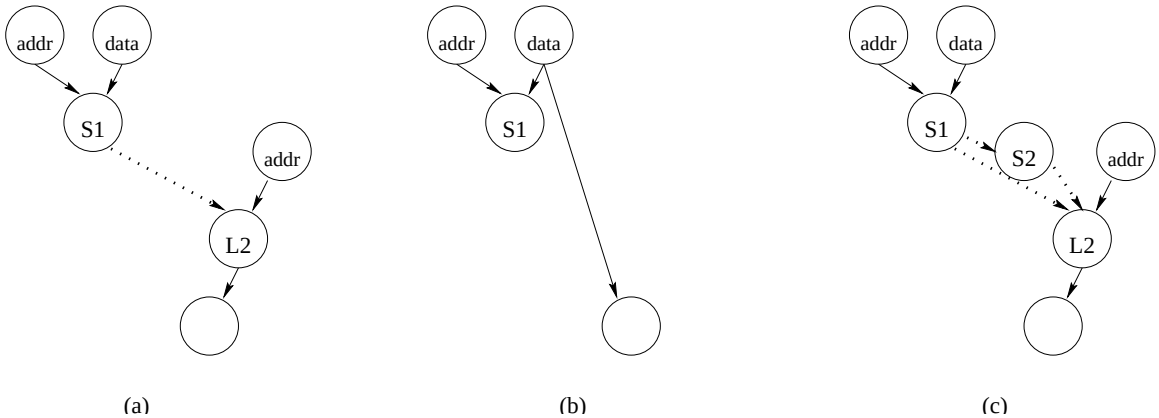


Figure 4.11: Redundant load removal using an earlier load. L1 and L2 are guaranteed to access the same location each iteration. (a) before optimization (b) after optimization (c) situation where optimization cannot be applied because of intervening store S.



(a) (b) (c)

Figure 4.12: Redundant load removal using an earlier store. S1 and L2 are guaranteed to access the same location each iteration. (a) before optimization (b) after optimization (c) situation where optimization cannot be applied because of intervening store S2.

stored by an earlier store S1 if all of the following conditions are true: (i) S1 and L2 definitely access the same location, (ii) there is no intervening store that might access that location, and (iii) S1 executes every iteration that L2's value is needed. Condition (iii) is equivalent to saying that S1 dominates L2. This optimization parallels Mahlke's *memory copy propagation* [Mah92].

Optimization: removing unnecessary stores A store S1 can safely be eliminated when a subsequent store S2 exists with the following conditions: (i) S1 and S2 definitely access the same location, (ii) there is no intervening load that might access that location, and (iii) S2 executes every iteration that S1 executes. Condition (iii) is true when S2 kernel-post-dominates S1 and in addition there are no intervening Exits between S1 and S2.

Optimization: removing redundant loads across iteration boundaries Array dependence analysis can determine when an access in one iteration is definitely to the same location as another access in a different iteration. In the particular case where a location is loaded one iteration and again in the next iteration, it is useful to instead retain a copy of the value rather than reloading it. One case leading to this situation is FIR filtering, a common digital signal processesing function that

computes the weighted sum of a sequence of samples from an input stream.

The compiler does not consider cases where successive loads are separated by distance $d > 1$ iterations, because these situations are more rare and because the net benefit deteriorates due to the area cost for retaining copies across multiple iterations. However, a chain of loads spanning multiple iterations, each separated by one iteration, will be successively converted so that all but the first load are eliminated.

In order for a load L2 to be removed by a load or store A1 in the previous iteration, all of the following requirements must be met: (i) A1 in iteration i is guaranteed to access the same location as load L2 in iteration $i+1$, (ii) A1 executes unconditionally, and (iii) there is no intervening store between A1 and L2.

When the conditions are satisfied, load L2 can be eliminated as follows. A new Hold node is created. All consumers of load L2 are modified to use the new Hold node's output. The Hold node's input is either the output of A1 if A1 is a load, or the node supplying the data value to A1 if A1 is a store. When A1 is a load, precedence edges from load L2 are transferred to A1, adjusting the distances accordingly. Specifically, if there is an incoming distance-1 precedence edge to load L2 from a store S, then a distance-0 edge must be added from store S to load A1 if one does not exist already. Finally, the inserted Hold node needs to be annotated with the address that load L2 would have accessed the first iteration; before kernel execution starts, the Hold node will be initialized with the value loaded from that address.

4.17 Removal of redundant precedence edges

The final step before writing out the DFG as an ASCII file is the removal of redundant precedence edges of all distances. This is done simply for efficiency reasons — it reduces the size of the ASCII file and also speeds subsequent synthesis. It does not affect the quality of the synthesis.

If there is a precedence edge E from node X to node Y, and there is a separate path P from X to Y of any edge types, and the cumulative iteration distance along P is less than or equal to the iteration distance of the edge E, then the precedence edge E can be eliminated. The path P may be

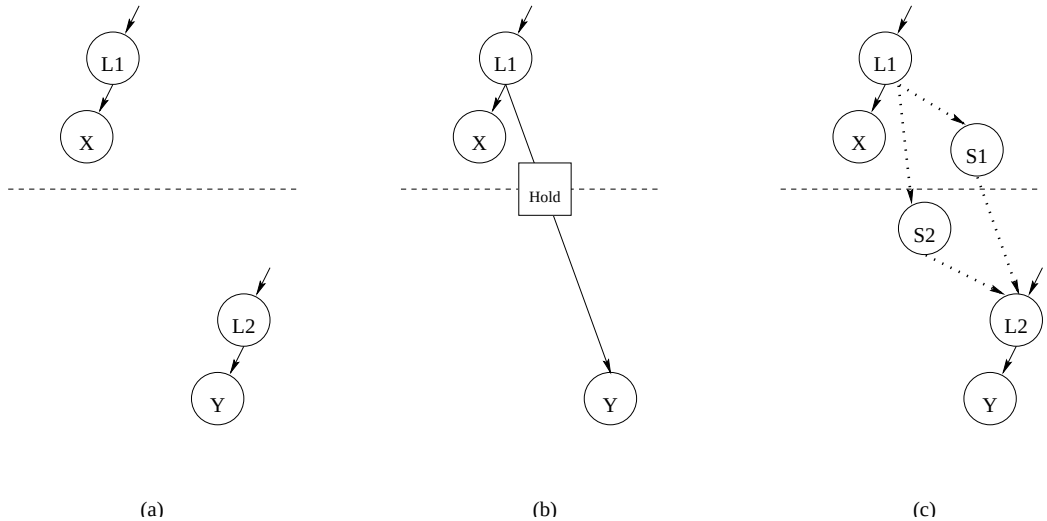


Figure 4.13: Example of redundant load removal across an iteration boundary. (a) before optimization (b) after optimization (c) situation where optimization cannot be applied (the existence of either S1 or S2 will inhibit the optimization).

simply a data edge from X to Y.

This makes the assumption that the required latency implied by the X to Y precedence edge is adequately enforced by the existence of any other path. While true for hardware synthesis targeting the Garp array, this might not be true for another target platform and back-end synthesis strategy.

It is important that this step comes last, in particular, after all optimizations that might remove a DFG node or edge. Otherwise, if a node or edge along the path P gets eliminated, then nodes A and B may be left unordered in the graph when in fact they should be. In other words, it would turn out that the eliminated A-to-B precedence edge is no longer redundant after the optimization.

Chapter 5

Utilizing Garp Memory Queues

Many of the loops that are good candidates for acceleration operate on contiguous streams of data. The memory accesses for such streams can often be fully overlapped with computation by buffering and reading ahead and/or writing behind. This buffering activity can be implemented as needed in Garp's array, but in the design of Garp [Hau00], it was felt better to provide dedicated hardware for this common task and thus free up more array resources.

Garp has three *memory queues* that support sequential streams. The queues are initialized by the main processor with a starting address and data size before array execution is started. From the array's perspective, queue accesses are like other memory accesses except that the array is not responsible for providing an address, reducing configuration size. Read response is also quicker since the data is already waiting in the queue buffer. And unlike regular memory accesses, which are limited to one per cycle due to the single address bus, all three queues can perform accesses every clock cycle over three independent memory data buses. The fourth memory data bus can still be used for a regular memory access, so that four accesses per cycle can be sustained, but only when three of them are queue accesses. Each queue can optionally be configured as non-cache-allocating so that streaming data used only once does not pollute the cache. A queue access can be predicated, so that it only executes during some of the loop's iterations.

The compiler tries to find as many memory queue uses as possible. A memory access must meet a number of conditions for conversion to a queue access to be legal. The first necessary

condition is unit stride: the address increment must match the data size, whether it be one, two, or four bytes. Secondly, although the increment and access do not have to both occur every iteration, the conditions (predicates) must be compatible. Finally, no ordering constraints as indicated by precedence edges in the DFG may be violated, since use of a queue effectively moves the actual memory access to a point earlier or later in time. The next sections detail the compiler's analysis to determine if an access meets these conditions.

5.1 Stride analysis

To determine when the address inputs of memory accesses have constant stride, `garpcc` performs *stride analysis*. The information derived from stride analysis is used by other DFG optimizations as well: induction variable recognition and strength reduction of multiplications involving induction variables.

The results of this analysis are, for each DFG node,

- Start: the value of this node during the first iteration. In general this is an expression involving constants and variables.
- Stride: the increment amount per iteration; this value must be a (compile-time) constant
- Predicate: if the increment is conditional, this value determines each iteration whether the increment is performed or not. If the stride is zero, the predicate is irrelevant.

Many nodes will not have a guaranteed constant increment each iteration; such nodes simply have "undefined" stride information.

The algorithm used is similar to the analysis used to detect induction variables [ASU86], although this case is simpler due to single reaching definition form of the DFG. The algorithm here also different in that it is extended to consider predicated (conditional) increments.

The algorithm for finding the stride information first finds strides for constants, loop-constant Inputs, and incrementing cycles in the DFG. It then finds additional information derived from these base cases.

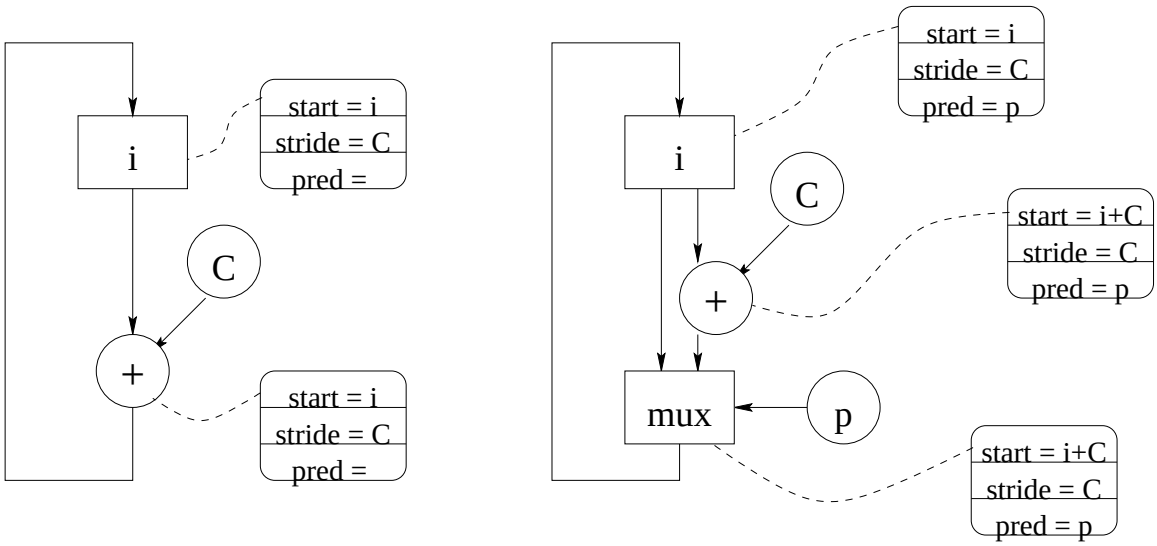


Figure 5.1: (a) simple incrementing cycle. (b) predicated incrementing cycle.

Finding trivial stride information Stride information is first added to Constant nodes and Input nodes. Their start value is simply their value; their stride is zero; and their predicate is irrelevant.

Finding incrementing cycles Incrementing cycles necessarily include a Hold node. In a simple incrementing cycle, the Hold is updated with its own value plus or minus a constant each iteration. In a predicated incrementing cycle, it is updated with either its own value, or its own value plus/minus a constant, selected by a mux. Stride information is added to each node in the cycle as indicated in **Figure 5.1**. Note that the stride information at the output of the mux in a predicated increment cycle assumes that the predicate is TRUE and is incorrect when the predicate is false.

Finding derived strides The next step is to derive stride information for other nodes, building from the stride information for increment cycles, Constants, and Inputs. One forward pass through the DFG nodes in topological order is sufficient to find all derived information.

When a node is examined, first its children are checked. If all of a node's children have valid stride information, then the node's operation is checked to see whether it combines the

Operator	Constraints	Derived Stride	Derived Start
neg	none	$-\ stride_1$	$-\ start_1$
add	none	$stride_1$ + $stride_2$	$start_1$ + $start_2$
sub	none	$stride_1$ - $stride_2$	$start_1$ - $start_2$
left shift	op_2 is a constant	$stride_1 << op_2$	$start_1 << start_2$
right shift	op_2 is a constant *and* $stride_1$ is evenly divisible by 2^{op_2}	$stride_1 >> op_2$	$start_1 >> start_2$
multiply	op_1 is a constant	$op_1 \times stride_2$	$op_1 \times start_2$
multiply	op_2 is a constant	$stride_1 \times op_2$	$start_1 \times op_2$
any simple unary operator	the operand stride is zero	0	$[operator]start_1$
any simple binary operator	both operand strides are zero	0	$start_1[operator]start_2$

Table 5.1: Rules for deriving stride information.

operands in a way that constant stride is preserved. If so, the appropriate stride information is attached to the node. Table 5.1 summarizes the rules used for combining stride information for different operators. Extending the rules for predicated strides is straightforward: either both operands must have the same predicate, or one can have a predicate and the other must have zero stride. If an operand has a predicate, the result has that predicate.

If any child has "undefined" stride information, or if the node does not match any rule's opcode and constraints, then the node is assigned "undefined" stride information.

The last two rules handle kernel-invariant expressions. When kernel-invariant expression optimization has been performed, as is the default, such expressions would have already been eliminated from the kernel.

Divide operators need not be considered since even division by constant divisors are considered infeasible and thus cannot occur in the DFG. However divisions of unsigned values by positive power-of-two constants will have been strength-reduced to arithmetic right shifts, which are both feasible and amenable to stride analysis. If the Garp synthesis flow did support constant divides, stride analysis could only be applied if the operands are unsigned: because division utilizes round-towards-zero semantics on the MIPS/Garp platform, there is an anomalous case if the signed dividend value crosses from negative to positive or vice versa. This anomalous case cannot utilize

the queues. In contrast, arithmetic right shift on signed values always rounds towards negative and thus exhibits no anomalous case crossing zero.

Although Garp queues can only be used for accesses with forward unit stride, this analysis is general in that it identifies constant strides of all values, including non-unit and backwards (negative) strides. Information concerning occurrences of unit and non-unit strides in kernels extracted from the benchmark suite is presented in Section 5.6.

5.2 Compatible predicates between increments and accesses

A Garp queue access performs as a unit both a memory access and a unit post-increment of the address associated with that queue. When a queue access is predicated, a single predicate input inhibits or enables both the access and increment. For an access to be converted to a queue access, the conditions under which the increment is performed must be compatible with the conditions under which the access is performed. To check this, the derived stride predicate is compared with the predicate guarding the access' originating basic block.

In the case of stores, the increment predicate and the access predicate must be identical. Either both must be unconditional (TRUE), or they must be equivalent Boolean expressions of branch conditions. The compiler simply tests if both predicates originate from the same DFG node; because common subexpression elimination and other simplifications have been performed, this catches practically all cases where predicates are equivalent.

This restriction on identical predicates for stores means that many common cases cannot directly use queue stores. The simplest example is

```
for (i=0; i<N; i++) {
    if (i & 5) a[i] = -1;
}
```

There is no way for the queue store to increment the address without writing a value. However, the above loop could be transformed through the addition of a queue load to enable the use of a queue store, as shown below.

```
for (i=0; i<N; i++) {
```

```
        tmp = a[i];
        if (i & 5) tmp = -1;
        a[i] = tmp;
    }
```

With this read-modify-write approach, both the load and the store can utilize queues. However, there is little benefit to such a transformation. The original loop can be pipelined to just one cycle per iteration even using a non-queue store. Also, the transformed version increases memory traffic and also has increased startup overhead for initializing the queues. Thus such a transformation was not implemented.

More latitude in the predicate relationship is allowed for loads. This is fortunate since loads derive greater benefits from queue usage than do stores. Consider the case below where there can be some iterations in which the increment is performed but the load is not in the original non-speculative context.

```
for (i=0; i<N; i++) {
    if (i & 5) sum += a[i];
}
```

This case can utilize a queue load. In those iterations where load would not be performed in software, the queue load is performed anyway; the queue is advanced and the new value is actually moved into the array, but goes unused that iteration. Memory traffic is not increased over the non-queue case which would speculatively execute all loads as well.

In general, as currently supported by `garpcc`, a queue load can be used when the load is performed in a subset of the iterations in which the address is incremented. If the increment has an associated predicate, then that is the predicate attached to the queue load. The CFG is used to determine this subset property. For it to be true, either the base increment must be unconditional, or its home basic block must kernel-dominate or kernel-postdominate the home base block of the original load instruction. Otherwise, there could be an iteration in which the load is performed but the increment is not.

In fact, if all details of the Garp architecture were exploited, there would be no restriction on the relationship between the increment condition and the access condition for loads to use queues.

In iterations where the load access but not the increment is performed, the correct behavior occurs because Garp array loads are "sticky" in the sense that the register retains the last value loaded from memory. Since the address has not changed since the previous load, the previously loaded value is exactly the desired value. As before, the increment's predicate is the one that would enable the queue load. Handling these cases was not implemented in the compiler because initial studies found them to be rare, and handling them would have added much complexity: it would be possible that the first access would occur before the first increment; this requires that the first value loaded would have to be placed in the queue load's target register before kernel execution begins. Also, it would be necessary to distinguish between conditional pre-increments and conditional post-increments.

5.3 Dependence considerations for queues

The Garp architecture does not guarantee coherency between a queue access and another queue or random access. For example, consider a queue load. The memory queue will start reading and buffering data from memory as soon as the queue is configured. If another store performed by the configuration changes one of those memory locations, it will not be reflected in the data delivered to the array, and incorrect behavior will result. Thus, for a load to legally be changed to a queue load, there must not be any previous store (by program order) that might change that memory location. Fortunately, it is straightforward to determine this from the DFG: if any store might change the memory location before that location is loaded, that store will have a precedence edge to the load. This leads to the following restriction:

> *A load can become a queue load only if it has no incoming precedence edges from any store.*

Similar consideration leads to the following restriction for queue stores:

> *A store can become a queue store only if it has no outgoing precedence edges to any store or load.*

Self edges are ignored during these checks (with arrays, they would have been eliminated anyway by accurate array dependence analysis). Thus a store's loop-carried precedence edge to itself will not prevent it from being converted to a queue store.

The above restrictions do not prevent two queues from accessing overlapping areas, even when one of them is a store. For example, in this simple loop, both the load and the store can be converted to queue accesses:

```
for (i=0; i<N; i++) {
    x[i] = x[i] + 1;
}
```

In this case the load only has an *outgoing* precedence edge to the store, indicating that the load in iteration i must be performed before the store of iteration i. This behavior is preserved by using queues, since an iteration's load will be performed far in advance of that iteration's store.

5.4 Conversion to queue access

Using the above stride, predicate, and dependence analyses, the compiler can identify those memory accesses that can legally use Garp's memory queues. A candidate must satisfy all three of the conditions to be a legal candidate.

If the number of legal candidate memory accesses is greater than the number of available queues (three), the compiler gives preference to loads over stores since performance benefits more from the elimination of load latency. If there are more candidate loads than available queues, `garpcc` makes an arbitrary selection of three.

Finally, each selected access is actually modified to become a queue access. This involves the following steps:

- The opcode on the node is changed to the queue version: a load becomes either a LoadQ (if unconditional) or a PLoadQ (if predicated), and similarly for a store.
- The address input to the access is disconnected, since a queue access does not need it. A subsequent dead node elimination pass will remove any nodes no longer needed.
- The information from the stride analysis—the Start address and the Stride increment—are attached to the queue access. This information is used to initialize the queue.

5.5 HW/SW interface when using queue accesses

If a kernel contains queue accesses, extra work must be done before and possibly after kernel execution. At entrance, for each queue, a *queue control record* (QCR) must be set up and then used for initialization. The QCR is a 160 bit structure located in memory. Most fields in this structure are known at compile time and can therefore be expressed as constant initialization data, reducing run-time costs. These fields include queue direction, data size, and the cache-allocate indicator. The only field that must be provided at run time is the starting address. This value is written to each QCR, then the special instruction `galqc` (Garp array load queue configuration) is invoked to initialize each queue using its QCR. These instructions are placed in the same basic block containing the instructions that move initial register values into the array.

At exit, after all live register values have been retrieved from the array, a `gareset` (Garp array reset) instruction must be executed if the kernel contained any queue stores. This instruction flushes the queue(s), so that any subsequent loads performed by the MIPS processor or array will see up-to-date values.

A real example of queue utilization is found in Appendix A.

5.6 Empirical data

5.6.1 Queue utilization

Since queue references resemble vector accesses, and vectorization of general purpose code is usually considered difficult (although see [Asa98]), it may be questioned whether this straightforward analysis can in fact find many queue uses in such code, particularly given the stride restrictions for using Garp queues. On the other hand, the hyperblock kernel formation approach should create more opportunities for queues since queues can be utilized even in cases when the entire loop is not vectorizable. Table 5.2 presents a histrogram over all successful loops extracted from the benchmarks, organized in two dimensions based on number of potential load and store queues. Of course at most three accesses can use queues even when the number of candidates is

Possible Queue Accesses							
	Stores						
Loads	0	1	2	3	5	6	10+
0	350 (55.8%)	59 (9.4%)	5 (0.8%)	0 (0.0%)	0 (0.0%)	3 (0.5%)	2 (0.3%)
1	142 (22.6%)	17 (2.7%)	0 (0.0%)	1 (0.2%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
2	27 (4.3%)	2 (0.3%)	1 (0.2%)	1 (0.2%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
3	5 (0.8%)	1 (0.2%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
4	7 (1.1%)	1 (0.2%)	0 (0.0%)	0 (0.0%)	1 (0.2%)	0 (0.0%)	0 (0.0%)
5	2 (0.3%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

Table 5.2: Number of kernels for each combination of potential queue loads and potential queue stores.

larger.

Over 43% of all kernels utilized at least one queue. There are few kernels that could utilize more than the available 3 queues. However, this data is skewed because only kernels successfully synthesized to fit into 32 rows are recorded. Kernels leading to datapaths larger than 32 rows could be expected to use more queues.

The slight blip in the histogram with large numbers of stores but no loads result from memory initialization loops, most commonly writing zeros. These indicate potential uses for Garp's multiword queue accesses; this was not implemented nor were complexities analyzed. This is further discussed later (Section 5.7).

Automated loop transforms could increase queue utilization in at least two ways. They could increase the number of memory accesses that have unit stride, and/or increase the average iteration count of the inner loop. No such transforms have been implemented in `garpcc`. Going further, Asanovic's thesis [Asa98] contends that many applications considered to be nonvectorizable can in fact be vectorized after appropriate restructuring, although such restructuring would be difficult to automate in a compiler. This class of transforms would be expected to increase the number of beneficial queue uses.

Stride	Number of accesses otherwise legal for queue
-1	14
2	18
3	2
4	4
5	5
8	1
10	2
12	1
20	4
26	1
256	2
512	2

Table 5.3: Breakdown by stride for accesses that meet dependence and predicate requirements but fail unit stride requirement.

5.6.2 Frequency of use of predicated queue accesses

Of the more than 350 queue uses found across all benchmarks, only 6 were predicated stores, and interstingly all of those were stores. Four occurred in '`go`' and two more occurred in '`gcc`'.

5.6.3 Non-unit stride occurrences

Table 5.3 shows occurrences of accesses with non-unit stride that otherwise meet the conditions for using Garp queues. Only accesses in the final version of successful kernels are considered.

This information might be used to make a decision about whether it is worthwhile to generalize Garp's queues to handle non-unit stride. But it also suggests compiler strategies to increase utilization of the queues. A negative unit stride occurs most often; depending on dependences in the loop, the compiler might be able to reverse the direction of the loop, resulting in a positive unit stride. For loads with strides of two or four, also common, the compiler could have the queues transfer two or four elements at a time but have the datapath ignore all but one. This would not increase traffic in most of the memory hierarchy since the minimal unit of transfer among the L1

and L2 caches and the queues is 128 bits—so loading every fourth 32 bit value moves 128 bits at a time anyway. Finally, non-unit strides might result from nested loops stepping through multi-dimensional arrays, in which case loop interchange might increase the number of accesses in the inner loop with unit stride. However, inspection showed that this last case rarely occurred; instead, non-unit strides were usually caused by a loop stepping through an array of structures.

5.7 Queue uses not recognized

The compiler falls well short of exploiting Garp memory queues to the full extent possible. As described in Section 5.2, there are certain combinations of differing increment and load predicates that are not transformed to use the queues.

Another limitation is that the algorithm for finding queue uses only considers one access per iteration. A case like

```
for (i=0,j=0; j<N; j++) {
    x = a[i++];
    y = a[i++];
    sum += x | y;
}
```

could utilize queue loads—in fact loading both values in a single cycle by utilizing two different memory buses—but the compiler does not look for such cases. A similar case involves an array of structures such as

```
struct { int foo; int bar; } a[N];
for (i=0; i<N; i++) {
    x = a[i].foo;
    y = a[i].bar;
    sum += x | y;
}
```

The following case is a bit simpler, and was encountered in ‘`gcc`’, but currently is not handled either. It is rejected because its analyzed stride is 2.

```
struct { int foo; int bar; } a[N];
for (i=0; i<N; i++) {
    x = a[i].foo;
```

```
        sum += x;
    }
```

A special case involves array initialization such as:

```
    for (i=0; j<n; j++) {
        a[i] = 0;
    }
```

Although such a kernel will be compiled to use a queue store executing at the rate of one iteration per cycle, much higher performance is possible. The stores could be performed four at a time by having the Garp array utilize all four data buses. However, as with loop unrolling or vector strip mining, additional complication would be added to deal with cases when the number of iterations is not evenly divisible by four.

Even greater performance is possible is possible if the array being initialized has elements smaller than 32 bits; by initializing four 8-bit elements via each 32-bit access, another 4-fold increase is possible, so that initializing 16 8-bit characters per cycle is possible. However, this attack would add even more complexity, this time arising from data alignment considerations.

5.8 Related work

Architecture support and automatic compilation for streamed memory accesses have been investigated in the context of normal instruction processors as well. The benefits are largely similar, as is the required compiler analysis. A good example is [MKW+98, McK95]. However, that work did not detail what if any coherence was provided by hardware, and thus whether their compiler needs to factor in dependence constraints and perform supporting analysis. Also, there was no mention of support for conditional increments and conditional accesses.

There is also a similarity between Garp queue utilization and vectorization of loads and stores [Wol89]. Similar dependence analysis is performed, and in both cases the targeted loads and stores cannot be part of a non-trivial dependence cycle. However, with `garpcc`, other parts of the loop including non-queue memory accesses can be involved in general recurrences that would have precluded vectorization of the loop without further transformation.

Garp also differs from vector architectures in regards to which predication combinations are best supported. Loads are handled more similarly than stores. Garp and `garpcc` supports cases analogous to "merge" and "compress" loads in the Titan vector supercomputer [DHM+88].

Because the Garp queue store couples the store and the address increment, with a single predicate enabling or inhibiting both, it could be described as a "compress" store as in this example:

```
for (i=0,j=0; i<N; i++) {
    val = ....;
    if (val > 0) B[j++] = val;
}
```

This "compress" store is not supported by common vector architectures. More commonly supported by vector architectures is the "merge" store, which selectively overwrites some of the existing memory values as in:

```
for (i=0; i<N; i++) {
    val = ....;
    if (val < 0) B[i] = val;
}
```

The Garp architecture does not directly support a merge queue store, although as described above (Section 5.2), `garpcc` could potentially implement the above loop using a queue-load-modify-queue-store approach.

The Titan architecture further supports an "expanded" store where every source value is written to memory, but skipping over some locations based on the values of a Boolean mask vector.

Finally, Garp queues do not directly support either gather or scatter operations. Yet both can be efficiently implemented on the array by using a queue load to fetch the address stream which is then fed to a regular memory access. Both cases can be pipelined to 1 cycle initiation intervals neglecting cache miss stalls—but the regular memory access may indeed encounter a large number of cache misses.

Chapter 6

Kernel Pruning

During initial kernel formation (Chapter 3), paths containing infeasible operations or inner loops were necessarily excluded from the hyperblock forming the kernel. Now, additional pruning of the kernel is considered: the exclusion of more paths from the kernel in order to help the kernel fit in available resources and/or to boost performance. Pruning is performed at this point—after initial construction of the DFG—because the DFG can be used to estimate the benefits and costs of each possible prune. As additional pruning is performed, the DFG is updated to reflect those changes.

The *prune edge* is introduced as canonical means of identifying one or more basic blocks to be pruned as a unit. The concept of compatible prune groups is then introduced. Then efficient means of evaluating the costs and benefits of each prune edge or prune group are presented. Finally different algorithms for performing the best prunes with reasonable evaluation costs are presented.

Pruning improves the hardware kernel; it does not consider hardware vs. software performance. The decision whether to revert to software is performed only after all pruning is complete by comparing the total performance using the pruned kernel (including overhead and the time spent in software on excluded paths) against the performance of the original software loop.

6.1 Prune edges

In contrast to VLIW hyperblock formation, `garpcc` forms hyperblocks by a subtractive process rather than an additive process. Rather than starting with a single basic block or path and then expanding from it, `garpcc` starts with all feasible paths in the loop and then selectively removes (*prunes*) from among the remaining paths.

Pruning must preserve the golden invariant: *every hw block is part of one or more cycles of hw blocks, all of which include the hw kernel entry*. Thus a prune cannot simply remove an arbitrary set of basic blocks from the kernel. The concepts of prune blocks and prune edges are useful to define a legal set of basic blocks to remove from the hyperblock.

A *prune block* is a basic block with two outgoing both of which are contained in the hyperblock; both of those edges are valid *prune edges*. The *prune victim set* associated with a prune edge is the set of basic blocks that become unreachable once the prune edge is redirected. If domination is directly extended to include CFG edges as well as basic blocks, a prune's victim set is exactly those basic blocks dominated by the prune edge. **Figure 6.1** provides an example of prune edges and associated prune victim sets. Basic blocks A' and C' are both prune blocks. A prune victim set may be empty as is C'-take in **Figure 6.1**; this is still a non-trivial prune, since such a prune may remove mux area and delay at the E' merge. A prune victim set might also contain multiple control paths as does A'-fall.

After the prune is applied, the basic block source of the prune edge becomes an Exit block, with only one outgoing CFG edge remaining in the hyperblock, the other exiting to software. The edge remaining in the hyperblock is no longer a prune edge.

Prune edge victim sets possess the nesting property: for two different prune edges in a kernel, either their victim sets are disjoint (containing no common basic blocks), or one victim set is a subset of the other. This relates to whether one prune edge dominates the other. If prune edge eX dominates prune edge eY, then eY's prune victims are a subset of eX's prune victims. On the other hand, if neither prune edge eX nor eY dominate each other, then their prune victim sets are disjoint.

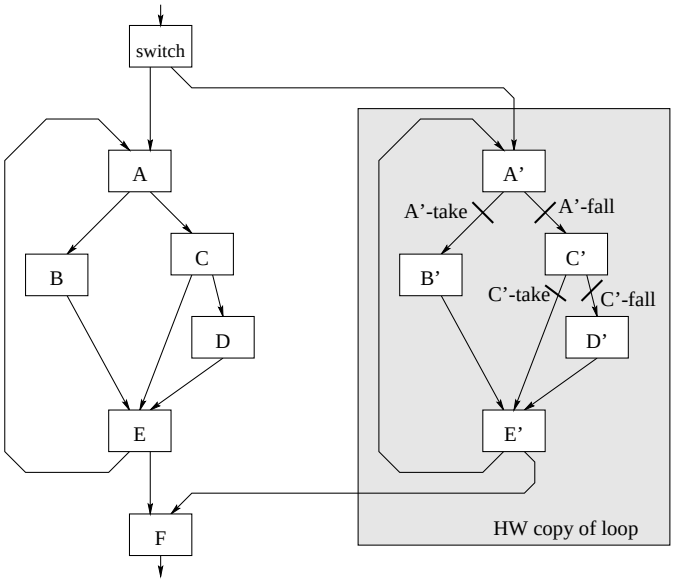


Figure 6.1: Prune edges in kernel.

6.1.1 Applying a prune

Once a prune is applied, it is not reversible. Applying a prune involves modifying the CFG, updating profiling counts in the CFG, and modifying the DFG. The application of a prune will also eliminate one or more other prunes from further consideration.

The CFG modification is straightforward. The prune edge is redirected to transfer control to the corresponding basic block in the software version of the loop. The prune victims in the hyperblock are eventually deleted since they are unreachable.

A prune alters execution counts downstream because paths taking an exit do not reenter the hyperblock. Updating counts helps subsequent prune steps make better decisions. The reduction in counts is shared evenly down stream, apportioning each value proportionally to the original execution numbers. This is done exactly as was done earlier to adjust for infeasible paths (Section 3.5). Path profiling [You98, BL96] could potentially be utilized to give more accurate updating of the profiling counts, but this has not been implemented or investigated.

After a prune, the DFG is updated to reflect the new form of the hyperblock. Currently

`garpcc` simply rebuilds the DFG from scratch, applying all optimizations, as this is a fairly inexpensive task. In theory it would be possible to edit the DFG and rerun those optimizations that might have new opportunities for application after the prune.

Once a prune edge is applied, its sibling prune edge is no longer valid since the sibling no longer meets the definition of a prune edge. In **Figure 6.1**, applying C'-fall removes C'-take from consideration; to subsequently remove the computation along C'-take, instead the prune A'-fall needs to be applied. Also, any prune edges contained within the applied prune edge's victim set are obviously no longer under consideration—in the example of **Figure 6.1**, applying A'-fall removes C'-take and C'-fall completely.

6.2 Compatible groups of prunes

Prune edges are defined so that any one can be applied individually, removing its associated prune victims, and still leave a legal kernel. However, it is useful to consider a *group* of prune edges to be evaluated or applied simultaneously. Not all groups of prune edges, though, can legally and usefully be applied together.

The simplest way to define compatibility is as follows: Two prune edges are compatible if they can be legally applied sequentially in either order. A group of prune edges is compatible when all members are pairwise compatible.

There are two situations where the application of one prune removes another from consideration, and thus they could not applied sequentially, nor are compatible pairwise.

In one case, consider the pair of prune edges, X-take and X-fall, originating at the same basic block X. Once one is applied, the other no longer meets the definition of a prune edge. It also makes sense that they should not be applied simultaneously: if both are applied, X would still be in the kernel but would have no successor remaining in the kernel, violating the golden invariant.

In the other case, consider when a prune edge eX is contained with in the prune victim set of another prune edge eY. Once eY is applied, eX no longer remains under consideration. This case occurs exactly when eY dominates eX.

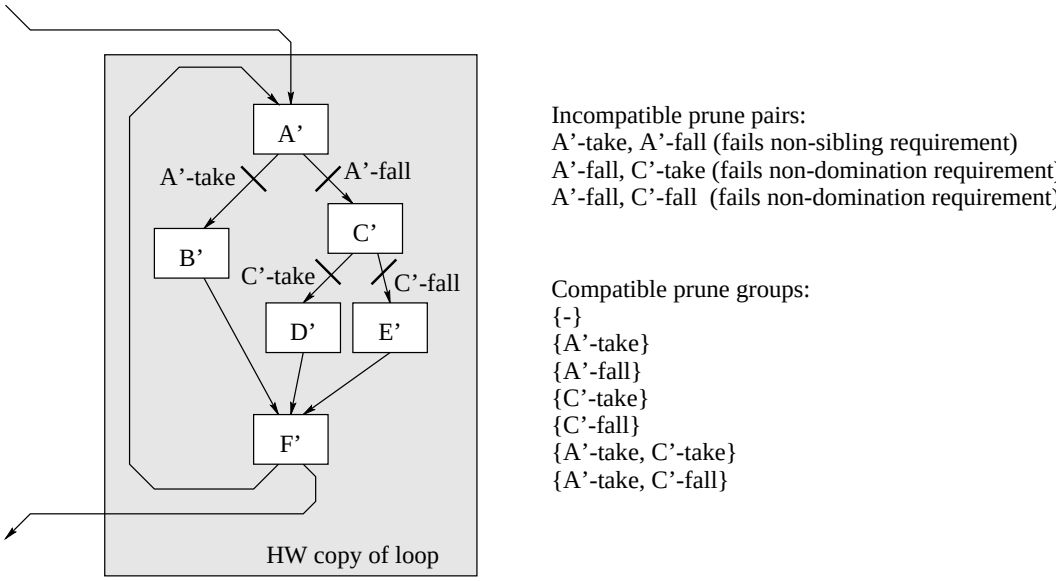


Figure 6.2: Compatible prune groups.

With these considerations, two prune edges are considered compatible when both:

- They do not originate at the same basic block (the *non-sibling* requirement), and
- Neither dominates the other (the *non-domination* requirement).

An upper bound on the number of compatible prune groups can be easily calculated by considering just the non-sibling requirement. Considering a kernel with n prune blocks, and thus $2n$ prune edges, there can be up to 3^n compatible prune groups (counting the empty group): for each prune block, one, or the other, or neither outgoing prune edge can be included. This upper bound considers the non-sibling requirement but not the non-domination requirement. In many cases the actual number of groups is much lower than this upper bound because of prune pairs that are incompatible due to the non-domination requirement.

In most cases, a prune group's victim set—the set of basic blocks that become unreachable—is the same as the union of the victim sets of the individual prunes. To complicate things, there are exceptional cases that must be considered, as in **Figure 6.3**. Basic block X is not

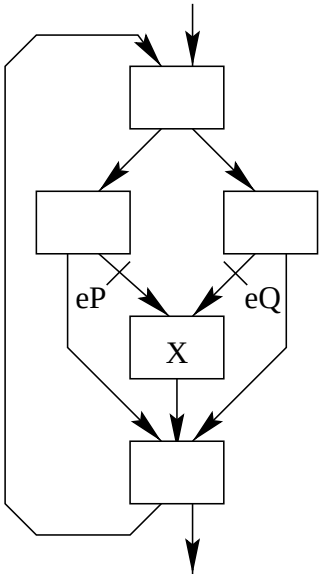


Figure 6.3: Example illustrating how a prune group's victim set can be larger than the union of the individual prunes' victim sets.

in the victim set of either of prune edges eP or eQ, since X is not dominated by either eP or eQ. However, it *is* in the victim set of the prune group {eP, eQ} since it becomes unreachable when both eP and eQ are applied. A CFG structure such as this cannot arise from normal C-language structured control statements—at least one `goto` statement is required—and is very unlikely to appear in human-written programs, although it may arise from the decomposition of switch/case statements.

6.3 Evaluation of prune benefits and costs

Applying a prune is irreversible in the compiler's infrastructure; but an estimate of the impact of applying a prune or prune group is required to make smart choices about applying prunes. This section describes how the impact of a prune is estimated without actually applying the prune.

Algorithms in following sections will consider either prune edges or prune groups. This section will describe evaluation in terms of prune groups, since a single prune edge can be treated as a prune group containing just that single prune edge.

The main mechanism is to simply add a “suppressed” flag to each basic block in the victim set of the prune edge or prune group under evaluation. Cross pointers between the basic blocks and corresponding DFG nodes make it straightforward in most cases to recognize suppressed DFG nodes. The estimation routines for area, delay, and exit costs are adapted to recognize suppressed basic blocks and DFG nodes and appropriately ignore them as described below.

The main complications in this process relate to the handling of DFG nodes not directly connected to any specific basic block: muxes, predicate logic, Input nodes, and Hold nodes. Another complication is the temporary insertion of new exit(s) in the DFG. These issues will be covered first.

6.3.1 Mux suppression and short-circuiting

Ordinary DFG nodes such as “add” can be completely suppressed and ignored when their owning basic block is suppressed; if that basic block is removed from the kernel, the addition would simply not be part of the resulting DFG. Mux nodes in the DFG are a special case since they do not strictly belong to any single basic block. They were inserted as necessary at merge points in the CFG, and thus involve at least two “incoming” basic blocks and at least one destination basic block. To complicate things further, the definitions that arrive via edges from the predecessor basic blocks are not necessarily contained in those basic blocks. These factors make determination of mux suppression more complicated than it first appears. Specifically, even if the DFG node producing a particular data input for a mux is not suppressed, the mux input may still be suppressed.

To aid the analysis, each mux records for each of the two data inputs the list of the one or more CFG edges by which that data value arrives. If there were more than two distinct data definitions, thereby requiring a tree of muxes, a mux input from another mux is considered to have inputs from all of the supplying mux’s CFG edges, both its true and false inputs.

Each mux data input is considered separately. A mux input is suppressed when all paths carrying that definition arrive via suppressed CFG edges. A CFG edge is considered suppressed when it is either an original prune edge, or it originates from a suppressed basic block.

If both of the mux’s inputs are suppressed, the mux itself is totally suppressed. If just one of the sources is suppressed, the mux is considered to be “short-circuited”—replaced by a direct

connection to the remaining non-suppressed data input; the mux's select input is disconnected in this case. It is important that a mux is not suppressed when just one of its data inputs is suppressed, or else the dependence between the remaining definition and the eventual uses would be broken.

Consider **Figure 6.4**. In particular note the mux for the variable '`i`'. The prune under consideration does not eliminate any actual definition of '`i`'. Correspondingly, neither of the DFG nodes that are data sources for the mux are eliminated. However, applying the prune *will* remove the path from the earlier definition, and thus the mux can be short-circuited during evaluation of that prune. Even if variable-to-variable copies had been retained as explicit copy DFG nodes, the implementation still could not be simplified to determine suppression of mux's input by checking if its source data DFG nodes were suppressed. Instead a dummy "passthrough" DFG node would need to be inserted for every variable that passes through a basic block unmodified. However, that approach would greatly increase the size of the DFG and also complicate implementation of many analysis and optimizations such as common subexpression elimination and constant folding optimizations.

The analysis did not go so far as to propagate the short circuiting of one mux to another when the affect of the first is to make the two data inputs of the second identical. This may add some minor inaccuracies in the estimates.

6.3.2 Predicate logic suppression and short-circuiting

Predicate logic suppression presents a case similar to mux suppression since predicate DFG nodes are not associated directly with any basic block in the CFG. Yet long dependence chains of predicate logic can have a significant impact on prune estimates. Thus predicate logic as well must be accurately suppressed for good prune estimates. This phase is performed after other types of nodes have been suppressed. Predicate logic with either all inputs or all outputs suppressed are suppressed themselves. An output to the control input of a mux that has been shortcircuited is considered to be a suppressed output. Suppression is propagated so that suppression of one predicate node can lead to suppression of others in either direction. Also, a binary predicate node (AND or OR) with one input source suppressed is "short-circuited" as in the mux case—as if its remaining

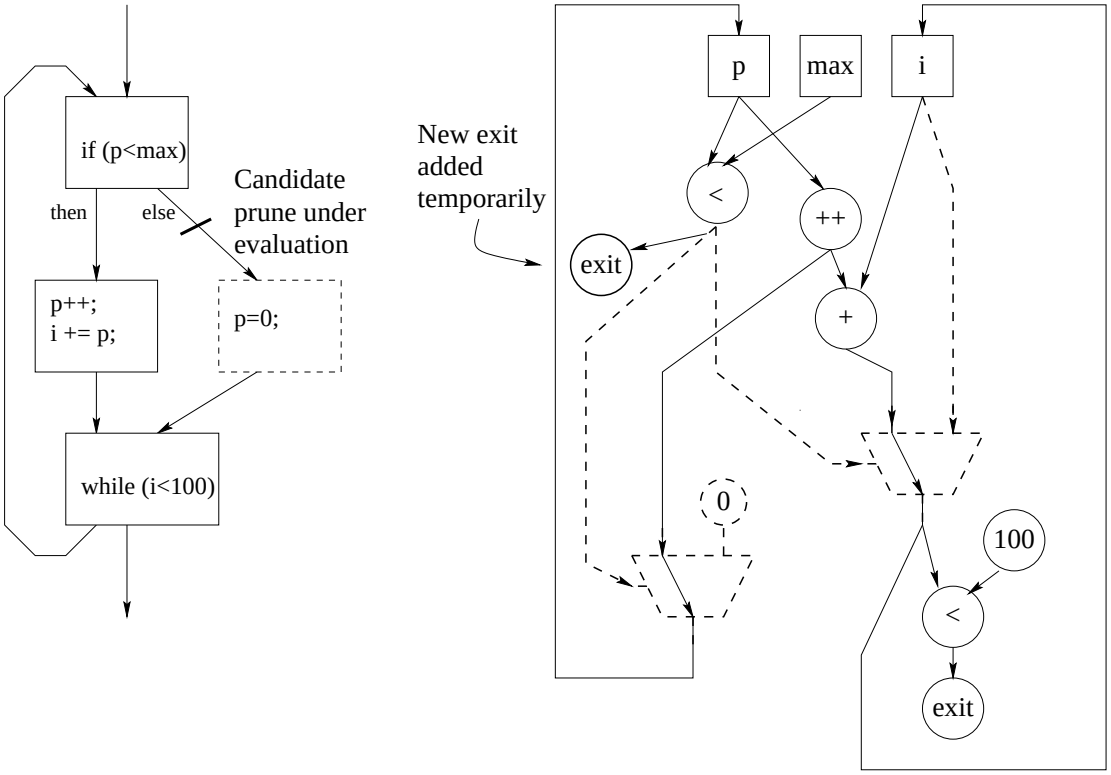


Figure 6.4: Evaluation of prune candidate.

input were connected directly to its consumers.

There is a special case for predicate logic attached to the condition that would control an exit if a prune under evaluation were applied. Were the prune applied and the exit constructed, no predicate logic would be connected to the condition because of the ‘partially resolved predicate’ approach taken by `garpcc`. Yet the condition source is not suppressed—its owning basic block is not suppressed and the condition will in fact be needed to control the exit. Thus the algorithm must make a special effort to recognize edges from such conditions and treat them as a suppressed source for predicate logic.

6.3.3 Input and Hold node suppression

Like the above cases, Input and Hold nodes do not strictly belong to a particular basic block. The case for an Input node is simple: it is considered suppressed when all of its consumers are suppressed.

The case for Hold node is slightly more complicated. It also is suppressed when all consumers—including exit nodes via liveness edges—are suppressed. Furthermore, the common case shown in **Figure 6.5** must be considered: one of the mux’s data inputs is suppressed and its other data input is a Hold node, which in turn is fed by the mux. This happens when the only modifications of the variable occur within the victim set of the prune under evaluation. In this situation, the Hold node’s value is in fact invariant, so it should treated as an Input node. If in addition all other consumers of both the mux and Hold outputs are suppressed, then both the mux and Hold are suppressed.

6.3.4 Exit insertion

The only direct modification of the DFG when evaluating prunes is the temporary addition of the new Exit node that would be inserted if the prune were applied. The temporary Exit node is removed immediately after the evaluation is complete.

Each pruned edge introduces a new hyperblock exit which translates to an Exit node in the DFG. In some cases such an Exit actually increases the delay. The new Exit node may introduce

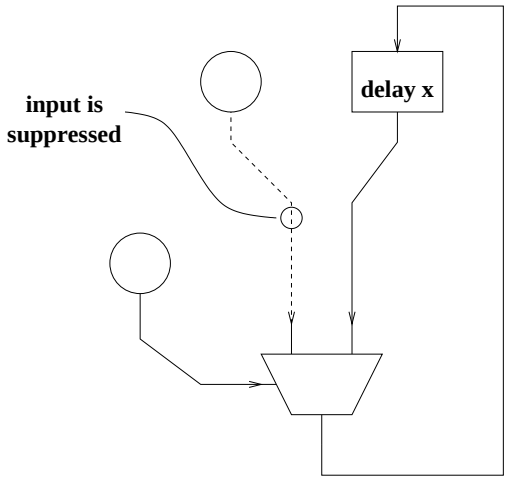


Figure 6.5: Case where loop-carried variable x becomes kernel invariant.

new dependence paths that increase the critical path/cycle of the kernel. Specifically, there will be a new precedence edge to the new Exit node from any exit or store on a CFG path from the loop entry to the new hyperblock exit. Similarly, there is a new precedence edge from the new Exit node to any exit or store downstream from it. There will also be new loop-carried precedence edges to/from the new exit node and *every* store and exit in the DFG. Finally, the data input to the new Exit node is approximated simply as the operation (typically a comparison) that controls the conditional branch of the basic block from where the prune edge originates. The exit insertion process does not attempt to construct the appropriate liveness edges to the temporary Exit node; this may cause some inaccuracy in the following estimates.

6.3.5 Estimating critical path/cycle

The estimated benefit is derived by comparing the critical paths/cycles before and after the prune. DFG nodes that would be eliminated by the prune are suppressed:

- Suppressed nodes owned are completely ignored by the critical path calculation. No dependence path passes through such a node.

- A short-circuited node are considered to have zero delay from its remaining data input to its output.

Even when a small region of the DFG elimination is eliminated, delay is recalculated over the entire graph. A smarter implementation would recognize when all of the nodes removed or short-circuited have slack, indicating that none are on the critical path/cycle; if this were recognized it would be known that the prune makes no effect on the critical path/cycle.

There is an indirect performance benefit from a prune that may affect the critical path/cycle. A prune typically reduces the overall size of the datapath; smaller datapaths could lead to some busses being shorter, reducing the routing delay and possibly the critical path. This benefit is not included in the prune heuristic, although it is ultimately realized in datapath synthesis.

6.3.6 Estimating area

Selective pruning of kernels will lead to smaller datapaths due to two effects. Firstly, the computation contained in pruned basic blocks is no longer implemented in the Garp array. Secondly, the muxes that had been required to merge the results from a path are no longer needed if the path is pruned. Muxes that are either suppressed or short-circuited have no estimated area. Finally, eliminated or short-circuited predicate logic also has no area, although this has a relatively minor impact.

Area benefit is calculated by simply totaling the individual estimated areas of the DFG nodes owned by basic blocks that would be pruned, in addition to muxes and predicate logic that can be short-circuited or eliminated.

There is another area contributor when pipelined execution is anticipated: the addition of registers in the datapath to synchronize the progress of a particular iteration's data (Chapter 8). This contribution is too important to ignore, but unfortunately it is much more expensive to estimate than the other contributions. Essentially the estimation routine must imitate the actions that will later be done in pipelined synthesis (Chapter 8): find the minimum initiation interval, perform modulo scheduling, and estimate the area of the inserted registers based on the slack at each DFG node's output after scheduling. Some simplifications are used: packing nodes to optimized modules is not

considered (each node is scheduled independently), and the modulo scheduling algorithm does not consider conflicts for the shared memory port.

6.3.7 Estimating exit overhead costs

Pruning the kernel will typically lead to a number of additional iterations taking kernel exits. These additional exits will incur exit penalty costs that can be broken down into three distinct categories.

The first cost is for the transfer of control and data to the MIPS core, and then back to the array at the start of the next iteration. The number of operands to be transferred back from the array depends on which exit is taken. Also, when there are multiple kernel exits, additional overhead results from determining which exit is taken. Queue flushing (at exit) and reinitialization (at reentry) are also added to this cost. However, only the removal of queue accesses by the prune are taken into account. Because the queue recognition algorithm is not re-run, the overhead from new queue accesses resulting from the prune is not considered. Approaches to reduce data transfer cost by retaining live state in the array and/or queues between kernel exit and re-entry have not been investigated.

Secondly, the time spent performing computation in the software "tail"—from a hyperblock exit to the end of the iteration—may be substantial. Typically it is much greater than the initiation interval using pipelined hardware execution. The software tail execution time from a given exit is calculated as a profile-weighted average of the software execution times of all paths from that exit to a backedge or a loop exit.

Finally, exiting and then reentering hardware disrupts the pipelined execution. The entire latency of the first iteration after hardware continuation is visible. This pipeline refilling cost is most significant in heavily-pipelined kernels.

It is important that the profiling counts are adjusted to take into account the effect of the prune under evaluation (Section 3.5). Otherwise some double counting of exits will occur, overestimating the associated overhead.

6.3.8 Estimating configuration overhead costs

When a prune results in a smaller configuration for a kernel, it will take less time to load the configuration when it is not in the configuration cache. While this effect is typically small compared to other effects, it can be important when the total kernel execution time is small, so that configuration time is relatively significant, and/or the prune makes a large difference in the size of the configuration.

The heuristic optimistically assumes that the configuration is loaded from main memory only the first time it is used and that subsequent uses will find it in the configuration cache. In reality some configurations may miss in the configuration cache a significant fraction of their executions. Thus the heuristic likely underestimates the true contribution of this factor.

6.3.9 Estimating performance: putting it all together

Judging the overall performance of the hardware kernel is framed in terms of the average number of cycles per iteration including amortized overhead costs. The calculation is

$$
\begin{aligned}
cyclesPerIteration \;=\; & II \\
& + configOvhd/iters_{total} \\
& + [\sum_{exit_i} (iters_{exit_i}/iters_{total})] \\
& \qquad \times [queueOvhd + mtgaOvhd + pipeStartup] \\
& + \sum_{exit_i} [(iters_{exit_i}/iters_{total}) \\
& \qquad \times (exitOvhd_{exit_i} + avgSwTail_{exit_i})]
\end{aligned}
$$

using the following definitions:

II initiation interval
$configOvhd$ assumed overhead for one-time configuration load from DRAM
$iters_{exit_i}$ number of iterations taking $exit_i$

$iters_{total}$ total number of iterations including both those taking exits and those remaining in the kernel

$queueOvhd$ overhead for queue setup and flushing

$mtgaOvhd$ overhead for moving initial values to array

$pipeStartup$ exposed latency for refilling the pipeline, equal to SL - II where SL is the schedule length

$exitOvhd_{exit_i}$ expected overhead for exiting at $exit_i$. This includes overhead for determining which exit was taken and retrieving values specific to that exit.

$avgSwTail_{exit_i}$ expected execution time in the software tail executed after taking $exit_i$

6.4 Pruning algorithms

Pruning has two functions: fitting the kernel to the available hardware and improving performance by excluding rarely-executed paths. A number of approaches have been investigated as described below.

For efficiency, prune edges very unlikely to be applied are removed from consideration before any of the below approaches are applied. Specifically, if a prune edge is traversed on average more than half of the iterations, it is eliminated from consideration—pruning such an edge would mean that the kernel would execute on average less than two iterations before taking an exit. However, empirical results showed that the number removed by this method was almost always insignificant in cases where it mattered—where the number of prune edges was large.

6.4.1 Iterative fit, iterative perf

This approach works in two phases. First it iteratively prunes edges until the kernel is estimated to fit into the available resources, picking the prune with the best (area benefit) / (additional exits) ratio. Then it iteratively applies individual prunes for as long as there is still a beneficial prune edge. A prune is considered beneficial when it is estimated to result in improved performance considering both the critical path/cycle improvement along with the increased exit costs.

This approach has some weaknesses regarding quality of results. Iterative fitting of the kernel to available resources is essentially a greedy approach to the knapsack problem [Pap94],

for which it is easy to construct examples where the optimal solution is not found. The iterative approach also has a particular weaknesses when improving performance. An example is where there are parallel long dependence chains originating from different rarely-taken CFG paths. The iterative approach will only consider eliminating one or the other, but not both. But eliminating just one will not improve the critical path/cycle, and thus neither will be applied. In this case the benefit will be recognized only by considering multiple prunes simultaneously. This motivates the enumerative approach.

6.4.2 Enumerative

This approach enumerates all compatible prune groups and estimates the area of each resulting kernel. For each group that results in a fitting kernel, it estimates critical path/cycle and exit costs and then chooses the group that results in greatest overall performance.

When a kernel has a large number of prune edges, there can be an enormous number of compatible prune groups, making the straightforward applications of this approach prohibitively expensive. Rather than completely revert to the iterative approach in such cases, this approach makes iterative applications of one prune edge at a time just until the remaining number of undecided prunes is small enough to feed to the enumerative algorithm. Each time one prune edge is applied, at least two and possibly many more prune edges are removed from consideration. The cutoff has been chosen at 20 prune edges and 2000 compatible prune groups; groups are counted only if the edge limit is not exceeded. As long as either is exceeded, iterative pruning continues. The “iterative-fit” heuristic is used to select a prune edge. In all observed cases the size of a kernel that exceeds the above edge/group limits far exceeds the estimated capacity of array sizes up to 64 rows, so the “fit” heuristic is appropriate.

The enumerative approach has its own weakness with regards to the quality of results. The iterative approach has an advantage due to the DFG rebuild each iteration. Because full DFG rebuild and optimization is done after each individual prune application, it may realize that the kernel fits after fewer prunes, while the enumerative approach may apply a larger group of prunes than necessary due to pessimistic estimates.

6.4.3 Enumerative fit, iterative perf:

This hybrid approach was initially considered when area estimation was thought to be much less expensive than II estimation. However, once it was realized that accurate area estimation depended on accurate pipeline area estimation, in turn depending on II estimation, any benefits of this approach were eliminated.

6.5 Empirical results

6.5.1 Comparison of approaches: number of evaluations

In cases where the number of compatible prune groups is much larger than the number of prune edges, the iterative approach can be much faster—especially when few prunes are applied. On the other hand, if there are a large number of *incompatible prune edges* (and thus a relatively small number of prune groups), and the number of applied prunes is large, the iterative approach can actually be more expensive, since each iteration it must rebuild and optimize the DFG and evaluate all remaining prune edges.

Table 6.1 segregates the kernels by number of prune edges in the initial (all feasible paths) kernel, and for each class, presents the average number of evaluations needed until kernel pruning completes. For the iterative approach, evaluations for fitting and performance improvement are broken out. For the enumerative approach, evaluations for pre-enumerative iterative reduction and then enumerative phases are broken out.

While the number of evaluations with the enumerative approach can be up to an order of magnitude greater than with the iterative approach, adjusting the 20 edge, 2000 group cutoff point could reduce the difference at the expense of some quality of results.

On the other hand, the enumerative approach required fewer DFG rebuilds. Measurements of the relative time required for an evaluation versus a rebuild was difficult because in most cases the time was less than the resolution of available timers. For very large kernels, the only ones for which times could be directly measured, it appeared that a DFG rebuild took between 5 and 10 times longer than a prune evaluation. For another measurement, execution of HWSSA was timed for the two

initial		Iterative pruning				Enumerative pruning			
edges	# krnls	fitting	perf	total	rebldS	pre	enum	total	rebldS
0	298	1.00	0.98	1.98	0.00	0.00	1.00	1.00	0.05
2	123	1.11	2.21	3.33	0.32	0.00	2.02	2.02	0.32
4	82	1.87	3.77	5.63	0.74	0.00	4.21	4.21	0.65
6	21	3.00	5.57	8.57	1.00	0.00	9.48	9.48	0.76
8	25	7.64	5.40	13.04	1.88	0.00	18.32	18.32	0.92
10	19	14.47	5.84	20.32	2.95	0.00	40.53	40.53	1.00
12	11	11.45	6.09	17.55	1.45	0.00	102.73	102.73	0.73
14	10	27.60	6.20	33.80	3.70	0.00	228.40	228.40	1.00
16	8	49.50	3.25	52.75	5.00	1.88	574.00	575.88	1.12
18	6	42.83	7.67	50.50	4.50	5.67	852.00	857.67	1.33
20	2	38.50	12.00	50.50	4.50	9.50	897.00	906.50	1.00
24	3	82.00	4.00	86.00	8.33	36.00	656.67	692.67	3.00
26	1	51.00	18.00	69.00	5.00	21.00	1024.00	1045.00	2.00
28	1	29.00	1.00	30.00	1.00	28.00	1.00	29.00	1.00
30	1	123.00	0.00	123.00	9.00	77.00	1152.00	1229.00	6.00
32	3	145.00	7.67	152.67	8.33	114.00	868.00	982.00	5.67
34	3	46.67	3.00	49.67	2.00	33.00	331.67	364.67	1.33
36	1	116.00	8.00	124.00	5.00	83.00	896.00	979.00	4.00
38	1	68.00	12.00	80.00	6.00	32.00	512.00	544.00	2.00
40	2	155.00	8.50	163.50	7.50	126.00	384.00	510.00	5.50
50	1	175.00	17.00	192.00	10.00	122.00	1024.00	1146.00	5.00
52	1	295.00	1.00	296.00	18.00	194.00	1024.00	1218.00	9.00
60	1	211.00	1.00	212.00	11.00	160.00	512.00	672.00	6.00
76	1	655.00	0.00	655.00	18.00	610.00	972.00	1582.00	15.00
90	1	309.00	3.00	312.00	9.00	282.00	1768.00	2050.00	8.00
124	1	1113.00	5.00	1118.00	21.00	1084.00	360.00	1444.00	19.00

Table 6.1: Numbers of evaluations and DFG rebuilds required for different prune strategies.

Category	Number of kernels
Same successful kernel	405
Only iterative successful	3
Only enumerative successful	8
Both successful, iterative better	14
Both successful, enumerative better	20

Table 6.2: Comparison of iterative and enumerative prune approaches

approaches using an input file `g23.c`, which contained many initially large kernels that required much pruning. The iterative pruning approach took 10.3 seconds (including 680 prune evaluations and 47 kernel rebuilds) while the enumerative approach required 91.9 seconds (including 6949 prune evaluations and 29 kernel rebuilds).

6.5.2 Comparison of approaches: quality of results

Comparing different pruning strategies cannot be done simply by comparing overall application execution times when compiled using the different strategies. That is because often there are counter-effects, where what should be a better prune in facts hurts overall performance. A better-pruned kernel may come closer to exploiting all of the available resources (according to estimation), but then may fall victim to inaccurate area estimation and end up being too large after actual synthesis. In this case the loop would completely revert to software execution. Another possibility is that better pruning might lead to more successful and beneficial kernels, which could in turn lead to configuration cache thrashing among the larger set of kernels.

Therefore a local metric of the effectiveness of the different pruning approaches is used: the estimated cycles per iteration including amortized overhead.

The results are presented in Table 6.2. The majority of kernels receive the same pruning under both iterative and enumerative approaches. When the kernel for one approach is missing, that means that its estimated cycles per iteration was worse than software. In those that differed, the enumerative approach provided a distinct but not overwhelming advantage.

	Reasons prunes applied							
Benchmark	(none)	infeas	fit	infeas, fit	perf	infeas, perf	fit, perf	infeas, fit, perf
pegwit	12	0	0	0	2	0	0	0
m88ksim	15	1	0	1	2	0	0	0
cpp	22	0	0	6	5	2	0	11
perl	5	1	0	1	2	3	0	2
li	7	0	0	1	0	0	0	0
c99	10	0	0	0	1	1	0	0
go	61	17	0	17	29	6	1	2
cc1	110	31	0	13	28	15	4	5
vortex	33	1	0	0	12	0	2	2
gzip	18	0	0	1	0	0	1	0
compress	6	0	0	0	0	0	0	0
mpeg2decode	13	1	0	0	5	0	0	0
ijpeg	45	8	0	2	5	3	0	3
Totals	357	60	0	42	91	30	8	25
(percent)	58.2	9.8	0.0	6.9	14.8	4.9	1.3	4.1

Table 6.3: Number of kernels receiving each combination of prune types.

6.5.3 Frequency of application of prunes

Table 6.3 presents data regarding how many kernels are pruned for feasibility, fitting, performance, and all combinations thereof. These results are from compilation using the iterative-fit, iterative-perf approach. Only “successful” kernels are counted.

The table shows that a majority of kernels (58.2%) implemented the entire natural loop, excluding no paths. Yet, a significant fraction (21.8%) required the exclusion of infeasible operations. 6.0% needed pruning to fit in the available resources, and with 25.4% of the kernels, `garpcc` performed additional pruning attempting to improve performance.

6.6 Postponing the removal of infeasible operations

This section describes an investigation into an alternate flow for kernel formation. In the end it was not used.

An operation such as a multiplication presents a particular problem in the compiler flow. As appraised by initial kernel formation, a multiplication may be infeasible, requiring the removal of its containing basic block—and all paths through it—from the kernel. Yet subsequent removal of other paths in the kernel may result in either making the operation feasible, moving it to a different basic block outside of the kernel, or eliminating it altogether, any of which would allow the original owning basic block and paths through it to remain in the kernel.

Multiplication operations in particular will be considered to provide a concrete example. There are at least four scenarios when an originally infeasible multiplication becomes feasible or is removed from the kernel, allowing its owning basic block to remain in the kernel:

- **const-mult** The multiplication becomes feasible when one operand becomes constant. This situation is immediately recognized from the structure of the (rebuilt) DFG.
- **mux-invar** The multiplication can be eliminated when operands both become invariant. This was described in Subsection 4.15. This transformation could in fact eliminate any infeasible operation.
- **strength-red** The multiplication can be eliminated through strength reduction when one operand is or becomes invariant and the other is or becomes an induction variable. Recognition of this situation is described below.
- **mux-mult swap** (Not implemented) **Figure 6.6** illustrates another transform. By pushing the multiplication above a mux, two constant multiplications result, both of which become feasible. A check was added to `garpcc` to see if this situation ever occurred, and no cases were observed in any of the benchmarks studied, so the transform was not implemented.

Strength reduction Consider a product $m \times (s + i \times d)$ where m is kernel-invariant and $(s + i \times d)$ is an induction value that starts at s and increases by a constant amount d each iteration. This computation can be replaced by a new variable initialized by $m \times s$ and incremented by $m \times d$ each iteration. This new computation requires only an addition.

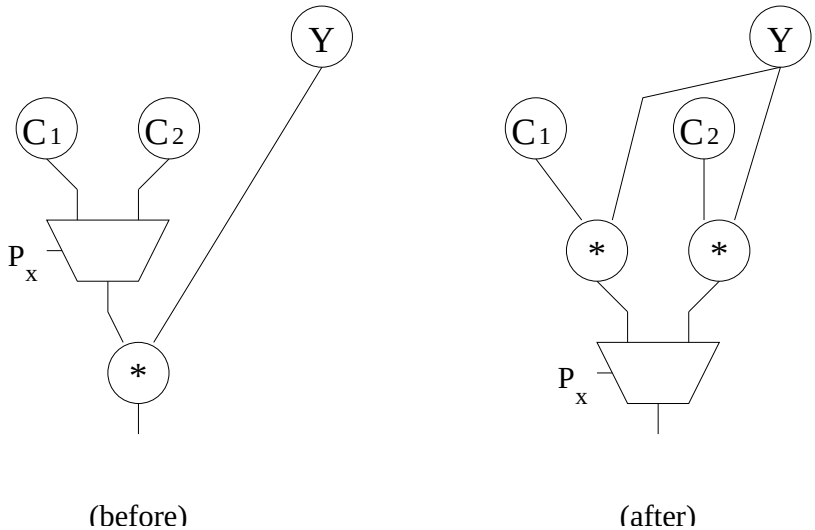


Figure 6.6: A transformation to eliminate an infeasible multiplication; C_1 and C_2 must be constants.

In fact this transformation would be able remove a multiplication even when the increment amount d is not a compile-time constant but is kernel-invariant. However, the stride analysis used to find DFG nodes producing values of the form $(s+i\times d)$, originally developed to recognize potential uses of Garp's memory queues (Chapter 5), only recognizes those cases where d is a constant, and has not been extended.

Integration with pruning It is quite likely that an operation originally classified as 'possibly feasible' will in fact remain infeasible after pruning has been performed and the DFG rebuilt. Thus a new final pruning step is necessary to remove additional basic blocks as necessary to remove any remaining infeasible operations. The new "iterative-feasible" pruning phase operates as follows:

1. Examine the DFG nodes until an infeasible node is found. If no infeasible DFG node is found, exit.

2. Find the infeasible DFG node's owning basic block BBo.

3. Set $BBcurr$ = BBo.

4. If $BBcurr$ is the loop entry, then the entire loop is infeasible; record such and exit.
5. If $BBcurr$ has an incoming CFG edge that is a valid prune edge, and that edge dominates BBo, goto step 7.
6. Set $BBcurr$ = any predecessor basic block of $BBcurr$. Goto step 4.
7. Prune the kernel at $BBcurr$'s incoming prune edge.
8. Rebuild and optimize the DFG.
9. Goto step 1.

This algorithm has the effect of applying the smallest single-edge prune that removes the infeasible DFG node. When there are multiple infeasible DFG nodes, it is smart to remove them iteratively, since the prune applied to remove one may result in another becoming feasible (or optimized out of the kernel).

The iterative-feasible phase is performed after either the iterative-fit / iterative-performance phases or after the enumerative fit+performance phase, whichever approach is being utilized. This has some obvious drawbacks since as currently implemented the fitting and performance pruning heuristics do not anticipate which additional prunes will be made due to remaining infeasible operations. The heuristics could be adjusted to lean towards prunes that remove a currently infeasible operation; the weakness is that the infeasible operation might eventually become feasible through the application of other prunes. The heuristic could also lean towards applying prunes that result in a non-pruned operation becoming feasible; however the required analysis for this would be complex—essentially requiring temporary application of suppression– and short-circuit–aware versions of constant propagation, strength reduction, and invariant expression elimination.

Preliminary results `Garpcc` was modified so that all multiplications were considered feasible until the final iterative-feasible pruning step. The iterative-feasible step was performed after all

other pruning was performed and the DFG had been rebuilt and reoptimized to reflect the earlier prunes.

Surprisingly this change in the complier flow was largely unsuccessful at removing originally infeasible operations from paths, at least considering only the final kernels. In all of the benchmarks, there were only three cases where an originally infeasible multiplication was removed to allow its owning basic block to be part of the final kernel. In one case both of the multiplication's operands became invariant. In the other two cases the strength reduction algorithm was applied. There were 9 cases with iterative fit/performance pruning and 15 cases with enumerative pruning where an infeasible multiplication remained so that additional pruning had to be applied. These cases may have been better off had the infeasible operations been removed immediately to give a more accurate picture for fitting and performance pruning. Because of this potential drawback, this approach of postponing the elimination of multiplications was deactivated when performing general experiments.

On the other hand, there were many more cases during early pruning iterations where originally infeasible multiplications became feasible. But those cases were on branches that were eventually pruned anyway for fitting or performance and so do not appear in the above numbers which consider only final kernels. This indicates that this approach may have more benefits when larger kernels become possible due to having greater amounts of reconfigurable resources. In addition, expanding the scope of this approach beyond multiplication may lead to greater benefits, although additional studies would need to be performed.

Also, the potential optimizations applied to remove a variable multiplication could be expanded. For example, this type of structure occured in the '`mpeg2decode`' benchmark (variable '`w`' is loop-invariant):

```
for (j=0; j<h; j++)
{
    jm1 = (j<1) ? 0 : j-1;
    ...
    tmp = 67*src[w*jm1];
    ...
}
```

If it were transformed to this case:

```
for (j=0; j<h; j++)
{
    wjm1 = (j<1) ? w*0 : w*(j-1);
    ...
    tmp = 67*src[wjm1];
    ...
}
```

strength reduction could be applied to the expression '`w*(j-1)`' and all remaining multiplications would be feasible. This transformation generalizes the interchange of muxes and multiplications to also consider potential new applications of strength reduction. In this particular case, however, it is likely that the kernel would be too big to fit on the array even if all variable multiplications were eliminated (although this problem could be addressed by another unimplemented transformation, loop fission).

6.7 Discussion

6.7.1 Cost of bad estimates

Bad estimates can lead to bad prunes in many ways. If a size estimate is too large, it may force an extra prune when in fact the kernel without that prune would fit. In this case more kernel exits than necessary are incurred, likely reducing performance. But the opposite case can be even worse; if a size estimate is optimistically small, it will later be found that the kernel cannot be synthesized to fit, and the loop reverts completely to software, losing any acceleration. These flaws could be corrected having a path from synthesis results back to prune selection, eliminating reliance on estimates.

6.7.2 Pre-DFG pruning

The pruning approach as described requires a DFG build of the initial kernel. In the case of very large initial kernels, even one initial build can be relatively expensive. Yet large initial kernels cannot be automatically ruled out for acceleration since large portions of them might eventually be

pruned yielding a beneficial kernel. In these cases it would makes sense to apply some pruning even during initial kernel formation. A very rough area estimate could be made directly from the software instructions, and when the result is many times larger than the available resources, extremely rare paths could be pruned. The goal would not be to produce a kernel that would fit, but simply to produce an initial kernel of reasonable size to allow DFG construction and post-DFG pruning to run faster. In other words, it would be smart to make the easy prunes earlier in the flow using a simpler, cheaper evaluation to save work downstream.

6.7.3 Alternative approach

An alternative, more straightforward approach would estimate the effect of each prune by duplicating the kernel, applying the prune, and then building and optimizing a new DFG for the duplicated pruned kernel. Call this the “build” approach.

The “suppression” approach to evaluating prunes described here makes the assumption that it estimates the effect of each prune group more cheaply than the “build” approach. While this is likely true, the complexity of the suppression approach was much greater than originally expected. If time allowed, the “build” approach would have been further investigated. It would give more accurate results of course since optimizations resulting from the prune group would be considered when comparing prune groups.

The “build” approach would be particularly useful with postponed of infeasible operations. When used with iterative pruning, it would be known when application of a prune edge would remove an infeasible operation from the remaining kernel. When used with enumerative pruning, prune groups that allowed an infeasible operation to remain in the kernel would be eliminated from consideration. Clearly enumerative consideration of prunes plus the “build” evaluation approach would give best results, but has not been implemented and so the cost of such in terms of compilation time can not be evaluated.

6.7.4 Viewing initial kernel formation as pruning

The removal of paths containing infeasible paths described in Section 3 can be reframed in terms of prunes as follows. The kernel initially is the entire natural loop, including infeasible blocks and inner loops. Then iteratively, each infeasible block or inner loop is removed by applying the minimal prune that contains it. The minimal prune is found by starting at the infeasible basic block (or inner loop entry block) and tracing backwards along any CFG path back towards to the loop entry block. The minimal prune edge is the first prune edge encountered while tracing backwards that dominates the infeasible block. If the loop entry block is reached before a dominating prune edge is encountered, then the infeasible block is unavoidable, and the entire loop is infeasible. Of course, a single prune may remove multiple infeasible blocks; only remaining infeasible blocks need be considered in following iterations. This formulation is equivalent to Section 3 only with structured control flow; it would prune *more* basic blocks than the approach of Section 3 in the example of **Figure 6.3**.

Chapter 7

Synthesis of Configurations

Since the Garp array design is novel, there was no existing synthesis flow for it. This was a mixed blessing. While it required the effort for construction of a complete synthesis flow, it allowed the exploration of alternative flows. Particularly, a new design approach became obvious: one that was compatible with a fully spatial approach; one that was compatible with the Garp array's fixed clock; and one that simultaneously optimized both grouping of operations into complex modules as well as relative placement of the modules.

The algorithm used here is specifically suited for a fully spatial implementation. The spatial approach allows merging of contiguous operations from the DFG into optimized compound modules. The approach is adapted from the instruction selection algorithm used for processors with complex instruction sets. In fact the generated datapath can be viewed as a a collection of static instantiations of complex instructions, interconnected by buses.

Motivation for the approach is best introduced by describing the array and examples of some possible modules.

7.1 Garp's reconfigurable array

The structure of the Garp array has many similarities to that of a field programmable gate array (FPGA). FPGAs consist of configurable logic blocks (CLBs), usually arranged in a 2-

dimensional grid, connected by a programmable interconnection network. Each CLB contains some amount of combinational logic whose input-output function depends on the configuration downloaded into it. Each CLB also typically contains one or more storage elements (registers, also called flip-flops) whose connection to the combinational logic also depends on the configuration. A desired combinational or sequential digital circuit can be realized by setting the configuration of the CLBs and the interconnection network.

Like an FPGA, the Garp array is a two-dimensional array of CLBs interconnected by programmable wiring. Like the processor, the array has a fixed global clock synchronizing all array operations. Unlike most uses of FPGAs, the speed of this clock remains constant for an implementation and is not adjusted for a particular array configuration.

It is natural, although not required, that 32-bit integer datapaths in the array be oriented so that operations such as addition span the central CLBs of each row and are stacked, connected to each other by vertical buses (**Figure 7.1**). There are vertical buses of different lengths, spanning 4 rows, 8 rows, 16 rows, etc. up to the full length of the array. The CLBs each contain individual 1-bit registers; the collection of these across the datapath portion of a row is often treated as a composite 32-bit array register. Each row has two such registers: the Z register, usually used to latch the output, and the D register, used to to hold an invariant value or to latch an input from a memory load or from a far module. The extra CLBs on each side of the datapath area are often useful for implementing controllers and computing Boolean data.

Along each row of CLBs, built-in carry chain hardware supports efficient additions, subtractions, and comparisons. Horizontal wiring channels between adjacent rows support shifts. These features together make multiplication by small constants fairly efficient as well. For example, multiplying a 32-bit variable by any 8-bit constant requires at most two rows and two cycles latency. Because of the flexibility of the CLBs, a single row can often implement a compound group of simple operations. For example, the C integer expressions `(a<<10)|(b&c)` and `(a-2*b+c)` can each be implemented in one array row with a latency of one cycle. The term “module” is used to describe such an implementation of one or more operations.

Memory buses provide the path into and out of the reconfigurable array. Garp’s array has

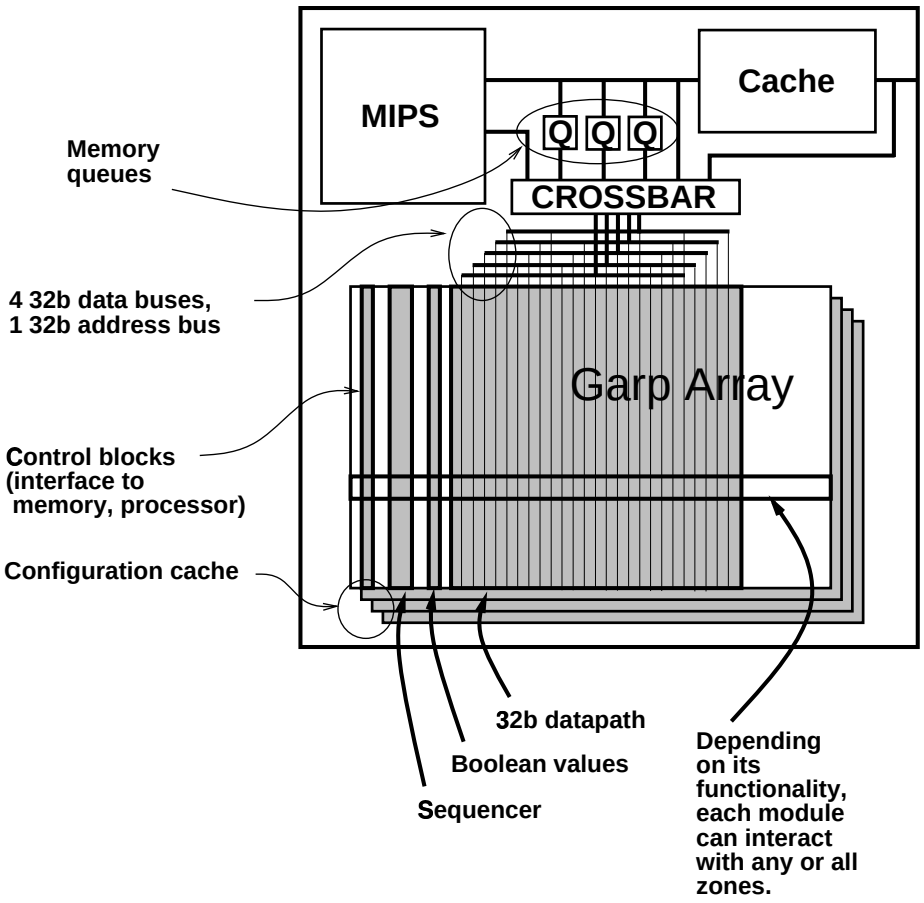


Figure 7.1: Garp array usage conventions.

four 32-bit data buses and one 32-bit address bus. While the array is idle, the processor can use the memory buses to load configurations or to transfer data between processor registers and array registers. While active, the array is master of the memory buses and uses them to access memory.

During execution, the reconfigurable array has access to the same memory system as the main processor, including all caches. To perform a random memory write, one row initiates the write and supplies the address, and another row provides the data. Random reads work similarly: once a row has initiated the read and supplied the address, the data is received into another row after a number of read latency cycles explicitly specified by the configuration. If the memory system cannot respond that quickly, the array stalls automatically, just as a regular processor instruction would interlock on a load that is delayed. At most, one row can initiate a random access each cycle since there is just one address bus. But the array can overlap accesses, initiating a new one every cycle.

With queue memory accesses, the array does not provide an address since the queue itself generates the address sequence after being initialized by the main processor. Typically, the data row initiates each queue access and either provides the next data element that cycle (for a store) or receives the next data element a cycle later (for a load). Since they don't compete for the address bus, multiple queue accesses can transpire in a single cycle.

Control blocks in the array's leftmost column, one per row, provide the row's control interface to the memory or processor. For example, depending on how a control block is configured, asserting one of its inputs might initiate a memory access, sending 32 bits of data from that row as the address. With an alternative configuration, asserting its input will halt array execution, thus acting as a loop exit. Any number of rows can be configured as loop exits, and they can all be evaluated the same cycle—if any is triggered, the array halts.

7.2 Examples of modules

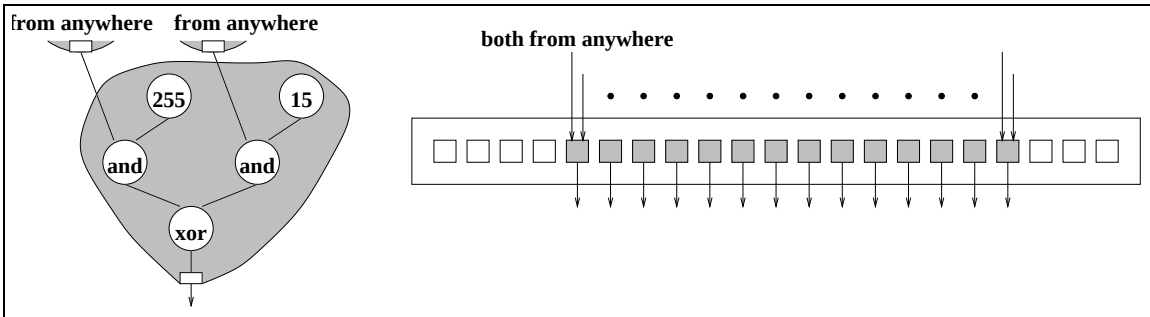
This section shows some examples of modules. A module is a minimal unit of placement and scheduling. Two modules can be linked in placement but not scheduling, such as when one must

be adjacent to another to facilitate a constant shift. Two modules can also be linked in scheduling but not placement, such as the two portions of a random memory access.

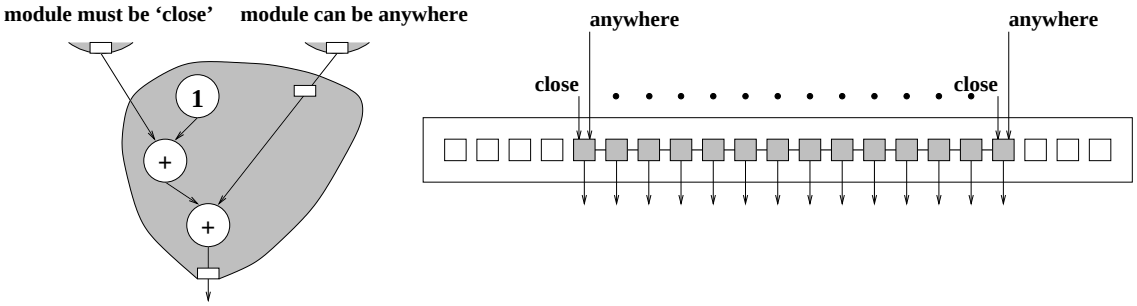
The modules are constructed so that no combinational delay between two registers exceeds one cycle. This is simplified by the fact that the Garp array timing model clearly states which combinations of transfer and computation have delays fitting into one cycle. In the model there are only two classifications of wire length: ‘short’ and ‘long’. Among vertical wires/buses, those spanning 4 or 8 rows are ‘short’, while longer ones are ‘long’. The path from a register, over a long wire, and through a simple logical function to another register takes one cycle. The path from a register, over a short wire, and through a carry chain operation to a register also takes one cycle. However, the path over a long wire followed by a carry chain computation exceeds one cycle and thus must be broken up by inserting an additional register between the long wire and the carry chain calculation.

The DFG is the same as that described in Chapter 4 except that random memory accesses are expanded to two nodes: the (possibly predicated) ‘memory’ node which initiates the access and supplies the address, plus either a ‘store’ node supplying the data to write or a ‘load’ node to receive a loaded value. The two halves of an access will become parts of two different modules.

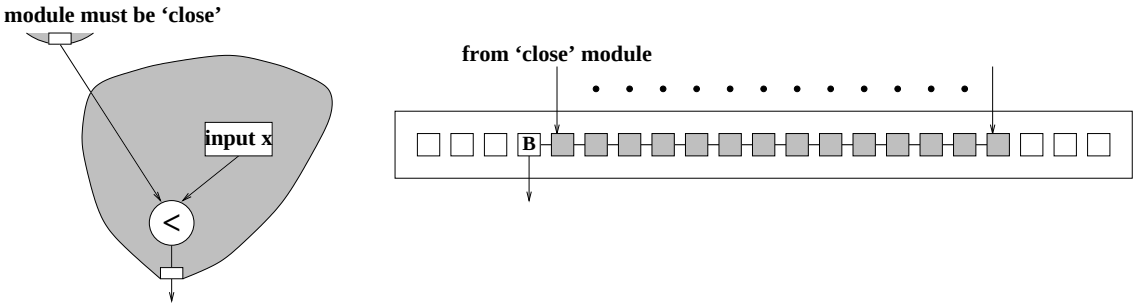
In the diagrams below, the left shows the DFG overlayed with a gray region indicating the region of nodes merged into a module. Small rectangles show register placement. On the right is a simplified depiction of how the module is implemented on the array. The usage conventions of **Figure 7.1** are followed; while this convention seems sensible for constructing a set of tileable modules, it is not the only one that is possible using the Garp array.



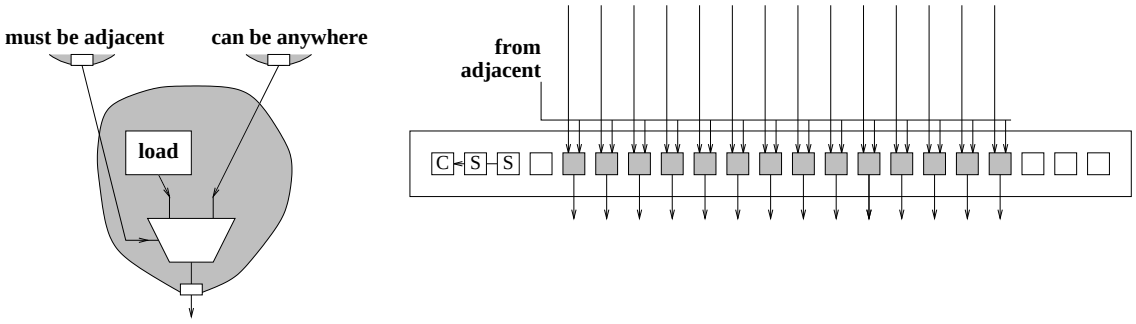
This module computes a bitwise-logical computation on two variable inputs and two constant inputs. It utilizes the CLBs’ combinational logic and output Z register.



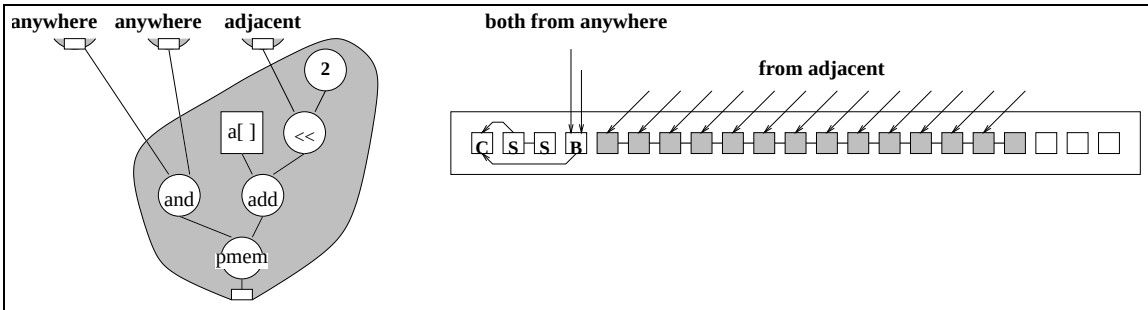
This module sums three values. Because the carry chain is used, inputs must either be buffered locally (latched in a register) or arrive over a short wire. The constant input is hardcoded into the configuration. One other is not buffered and therefore must arrive over a short wire from a close module. The final input is buffered using the row’s D register and thus can arrive from a far module; however this path incurs two cycles of latency from the source module’s output to this module’s output. A module cannot have two buffered inputs since there is just one D register.



This module performs a less-than function using the row’s carry chain. Because the carry chain is used, both inputs must either be registered locally or arrive over a short wire. One input is the kernel-invariant value of variable ‘x’; it is loaded into the row’s D register before execution starts. The other input must arrive over a short bus, and so the supplying module must be ‘close’ The result of the comparison is available in the Boolean column of the array (‘B’).



This module implements a multiplexor. One of the data inputs is the value from a load; the data is received into the row's D register when the sequencer ('S') triggers the row's control block ('C'). The other data input can arrive over a short or long bus. The module supplying be control input must be immediately above this module; the source module directly drives a global wire between the modules, used by each CLB in the mux to select the correct data input. For a case where the control input is not available from the adjacent module, the control input would need to be latched in the D register of the CLB in the Boolean column and then driven across a global wire, adding an extra cycle of latency.



This module initiates a predicated memory store, providing the address. Some of the predicate and address calculations are performed within the module. The data input must be produced in the adjacent module so that the shared wiring channel can be used to implement the constant shift. The sequencer ('S') ensures the memory access is initiated only during the correct cycle, while the calculated predicate ensures that it is initiated only during the appropriate iterations. When both occur to trigger the control block, the 32 bits in the module's output Z register are driven on the address bus to the memory system. At most one row should drive the address bus any given cycle.

7.3 Existing approaches to FPGA synthesis

Even though no automated design flow existed for the Garp array, it was worth considering existing techniques for FPGAs because of the similarities in their structure.

Computer-aided design (CAD) tools are indispensable in the realization of large circuits using FPGAs. Because FPGAs were originally developed with primarily random logic applications in mind, these tools typically perform their tasks at the level of individual gates or Boolean equations. These CAD tools perform at least three steps: technology mapping, which partitions the gates into groups that can be implemented in a single CLB; placement, which assigns each group to a specific CLB in the array; and routing, which assigns specific routing resources to form the appropriate connections between CLBs. The traditional CAD flow performs these tasks separately in the order described.

As the capacity of FPGAs has increased, they have been used more and more in datapath-intensive applications consisting primarily of multibit logical and arithmetic operations. The gate/CLB-level CAD flow, however, performs poorly with datapath designs.

There are several problems with the direct approach of first implementing each node with a datapath component, then flattening the datapath components to gates (discarding information about regularity) and feeding the resulting circuit to the gate-level design flow. Because the placement step often utilizes a pseudo-random simulated annealing approach, it is unlikely that an efficient bit-slice layout will be rediscovered (**Figure 7.2** a). The generated irregular layout leads to a difficult routing problem, resulting in long compile times and/or poor results. Also, flattening to gates leads to a much larger problem size—there are many times more gates than there are nodes in the DFG. Since many popular CAD algorithms have greater than linear complexity, this can lead to a dramatic increase in mapping and placement time. Also, once the circuit is flattened to gates, it is usually not possible to rediscover uses of specialized features of the CLB such as fast carry chain circuitry. Finally, the Garp array's interface with memory and main processor greatly complicate things: placement constraints would have to be added to ensure that the collective bits of some input or output value for address or data would line up along the appropriate columns of a row. This final

consideration alone makes the gate-level approach impractical.

A better approach for datapath circuits is to map each node to a predesigned, preplaced 'hard macro'. This approach can either use vendor's macro generators such as Xilinx's X-BLOX, as did PRISM-II [WAL+93], or build a new generator system such as MARGE [GG97] used by the NAPA-C compiler [GS98]. This approach can be fast, and it leads to a regular bit-slice layout. However, assuming modules with fixed layouts, no optimization across module boundaries is performed. This approach can lead to the underutilization of FPGA computation resources, especially with coarse-grained architectures (**Figure 7.2** b).

A hybrid synthesis approach might use hard macros for operations needing specific placement for interfacing, then use gate level optimization for regions of other Boolean and arithmetic operations. Yet the optimized regions would still suffer from irregularity as well as difficulty in exploiting special array features.

An academic research project, Koch's Structured Design Implementation [Koc96], used a novel approach to create optimized yet regular datapaths. After performing simple module selection and placement, it performs a module compaction step using standard gate-level tools to optimize one bit slice of the datapath and then tiles the results together. One limitation of this approach is that the module compaction step cannot handle specialized CLB features such as a fast carry chain, and thus does not attempt to merge modules that utilize such features. Another limitation is that only physically adjacent modules in the previously determined floorplan are considered for compaction.

7.4 The GAMA approach

The approach implemented in `garpcc`'s datapath mapping tool GAMA[CCDW98] has a number of advantages over these approaches. GAMA's main feature is that it is extremely fast; it does not flatten the modules to gates and so has a small problem size; furthermore, it utilizes a mapping algorithm that is linear in the number of nodes in the DFG. Because it operates directly at the module level, it can intelligently utilize datapath-level features of the reconfigurable fabric such as fast carry logic. Directly creating optimized, stackable modules also allows easy construction

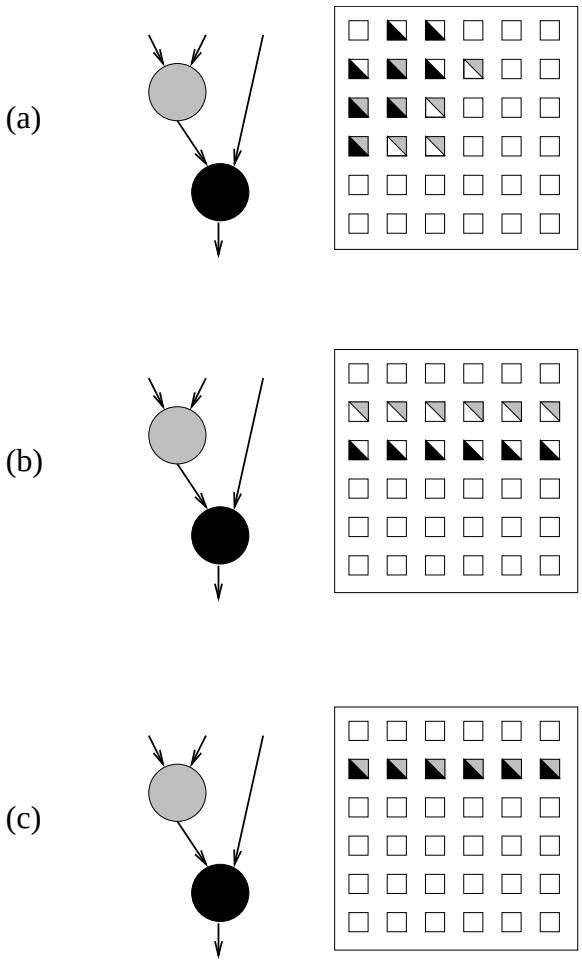


Figure 7.2: Different approaches to implementing datapaths. (a) Implementing each operation as basic gates and then feeding to traditional flow. Regularity is lost. (b) Implementing each operation as a hard macro. Computation resources are underutilized. (c) Ideal approach of merging operations while maintaining regularity. Merged module is ready to tile with other modules in bit-slice datapath.

of a regular bit-slice datapath. In a novel extension to module mapping, GAMA simultaneously considers linear module placement in the datapath in a way that preserves the linear time complexity of the algorithm. Knowing the relative placement of modules allows GAMA to accurately estimate the routing contribution to the overall delay along different paths and thus make better mapping decisions. Finally, this approach is flexible; it has been utilized in mapping to two different FPGA fabrics: the Garp chip's reconfigurable array as described here, and Xilinx 4000 series FPGAs as described further in [CCDW98]. Although both are based on 4-input lookup tables capable of implementing any 4-input Boolean function, these two arrays present very different mapping problems.

When mapping to the Garp array, GAMA generates a heavily registered circuit so that no register-to-register combinational path delay (including both logic and interconnect delays) exceeds one cycle of the Garp array's fixed-frequency clock. GAMA inserts registers as necessary to break a long interconnect/logic path into shorter paths. This is possible with Garp because GAMA performs placement simultaneously with mapping and can use the Garp array's simplified interconnect delay model to get accurate upper bounds on routing delays. Using these registers contributes no additional setup or hold delay in the Garp timing model and make subsequent pipelining easier.

The main operations performed by GAMA are outlined below.

Splitting into trees Since GAMA utilizes a tree covering algorithm that does not directly handle graphs containing cycles or nodes with fanout, the input DFG must be split into a forest of trees. Each tree is fed to the tree covering algorithm, and the results ultimately reconnected. Cycles are broken by removing loop carried edges. This produces a directed acyclic graph (DAG), which must be further split into trees. The simplest approach is to split the DAG at the output of each multiple fanout node (**Figure 7.3**(a)). GAMA also considers duplicating a shared subtree if it is small, since duplication can lead to faster and smaller mappings in some cases (**Figure 7.3**(b)). The decision to duplicate is based on an ad-hoc heuristic that considers the size of the subtree as well as the opcodes of the nodes around the split point. Note that even if each tree covering is optimal, the overall solution is likely not optimal using this approach. However, optimal covering of DAGs is

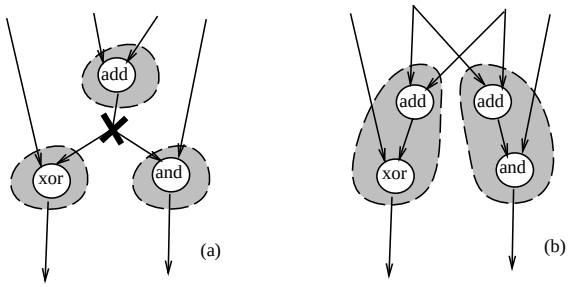


Figure 7.3: DAG splitting alternatives. (a) Force split (**X**) at every point of fanout. (b) Duplicate subtree; can lead to smaller and faster mappings.

NP-complete [AJU77] and so is not directly attempted.

Tree covering As has been shown, it is often possible to implement multiple nodes from the DFG together in a compound module that is much smaller and/or faster than if each were implemented separately. Typically a compound module consumes a single row of CLBs, but it could be of any size. When such compound modules exist, there may be many different ways that the DFG can be *covered* with module patterns from the library of possible modules. Although the number of possible coverings of a tree can be exponential in the number of nodes in the tree, a dynamic programming approach is used to find the best cover in linear time. This algorithm is the heart of GAMA and will be described further in the next section.

Each tree is passed to the tree covering algorithm separately. The trees are covered in topological order: a tree that produces a certain value must be covered before a tree that uses that value as an input at one of its leaves. The delay as calculated by the covering of the producing tree is used as the arrival time for the input to the consuming tree.

This phase adds annotations to the DFG indicating module boundaries, inter-module placement constraints, and other information needed for generating the modules correctly.

Post-covering rearrangement and optimizations A pass over the nodes after covering is used to perform some localized optimizations. Opportunities for these optimizations often arise at bound-

aries between different trees when they are reconnected after the covering. For example, an adder's input assumed to be far may turn out to be close, allowing the removal of the buffering of that input—eliminating a cycle of delay. These optimizations update the annotations on the DFG.

To increase opportunities for such optimizations, this phase first globally rearranges the modules after they have been locally placed by the tree covering algorithm. This allows layout possibilities that are not considered by the tree covering algorithm, such as intermingling the modules from different trees. However, placement constraints from the tree covering phase, such as adjacency of two modules, must be respected during the global rearrangement. The global rearrangement will also consider placing unrelated Boolean and datapath modules in the same row when it is easy to verify they do not interfere with each other.

Besides increasing opportunities for locality-based optimizations, module rearrangement also leads to a more routable datapath.

Module generation After modules and placement have been finalized, each specified module must actually be generated. A rich variety of functions can be implemented using a row of 4-input LUTs augmented with fast carry chain circuitry such as in the Garp array. It is therefore not feasible to simply instantiate each module by copying it from a static library, as the necessary library would contain tens of thousands of possible modules. Thus all modules are generated on demand. The generator, given a pattern of DFG nodes, annotations on the nodes, values of constant inputs, etc., creates the module.

7.5 Tree covering

GAMA uses a linear-time tree covering algorithm for finding the optimal mapping of the DFG nodes to simple and compound modules. The algorithm and underlying theory was originally developed for code generation in compilers [AG85], and was first used for the analogous problem of technology mapping in Boolean circuits by Keutzer in DAGON [Keu87]. GAMA utilizes `lburg`, a tool developed for the task of code generation in the `lcc` compiler [FH95]. Some modifications to `lburg` were necessary as described in Subsection 7.5.4. This modified version of `lburg` translates

a target-specific grammar representing the module library into the actual tree covering subroutine that gets compiled into GAMA.

7.5.1 Basic algorithm

For review, this section describes the basic tree covering algorithm. The algorithm uses dynamic programming, labeling the nodes in topological order from leaves to the root, combining previously calculated solutions to create new solutions at each node. The algorithm is given in pseudocode in **Figure 7.5**. The following definitions, illustrated in **Figure 7.4**, are useful in understanding the algorithm:

– The *pattern library* contains the patterns with which the input tree is to be covered. Each pattern is a graph of one or more nodes corresponding to a module that can implement that computation subgraph. Because of this correspondence, *module* and *pattern* are used interchangeably.

– A pattern P from the pattern library *matches* at node N in the input tree if, when P is overlayed on the input tree with the root of P aligned with N, the type of each node in P is the same as that of the corresponding node in the input tree. That is, the root node of P matches N, the left child of the root of P (if present) matches the left child of N, etc.

– The *fan-in nodes* for a pattern P that matches at node N are those nodes that are immediate predecessors of a node covered by P but are not covered by P themselves. The *fan-in trees* are the subtrees rooted at the fan-in nodes.

The best cover at node N is calculated as follows. Every pattern in the library is compared at node N to see if it matches. For each matching pattern P, the cost of the resulting cover is calculated by combining the cost of pattern P with the costs of the best covers at each fan-in node of P at N. The way the costs are combined can be unique for each pattern and is specified in the library. In general there will be multiple matches at node N and thus multiple covers. Only the best cover—that with the least cost—is retained, and the rest are discarded. The ‘cost’ contains separate fields for area and delay; Subsection 7.5.4 will discuss costs further.

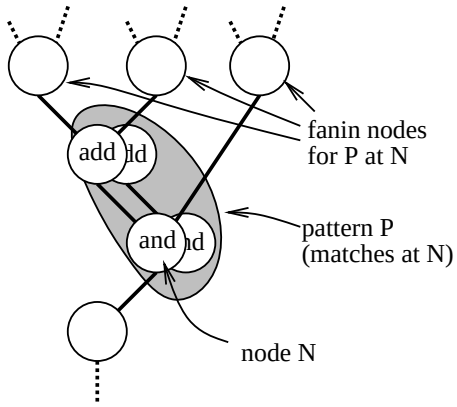


Figure 7.4: Definitions for tree covering.

When the root of the tree is finally labeled with its best cover, the best global covering has been found, although the information is distributed throughout the nodes in the tree. The patterns making up the best cover can be found by noting the pattern P recorded as the best match at the root, finding each fan-in node for P, and then recursively descending, finding the pattern that is the best match at each fan-in node, etc. This corresponds to the 'emit' phase of instruction selection [FH95], but instead of printing out assembly instructions, it adds annotations to nodes in the tree indicating module boundaries and other information needed to generate modules that interact with their neighbors correctly.

7.5.2 Complexity

While conceptually every pattern is checked to see if it matches at every node, the code produced by `lburg` is optimized so that in practice many fewer checks are actually made. Specifically, the covering routine first looks at the type of the node being covered, and then branches to a section of code that only checks for those patterns that have that same node type at their root. Typically the number of patterns that apply at any specific node type is a small fraction of the total. However, this is dependent upon the library itself, and in the worst case, all patterns need to be checked at a node. Thus, assuming that the pattern matching and cost evaluations are all of constant

```
function coverTree(N) {
    /*** postorder traversal of tree ***/
    foreach node K in children of N {
        status = coverTree(K);
        if (status == ERROR_NO_COVER) return(ERROR_NO_COVER);
    }
    /*** now label N ***/
    curBestCost = ∞;
    curBestMatch = null;
    foreach pattern P that matches at N {
        fanins[] = fanin nodes for P at N;
        forall i, faninCosts[i] = fanins[i].bestCost;
        C = P.costFunction(faninCosts[]);
        if (BETTER(C,curBestCost)) {
            curBestCost = C;
            curBestMatch = P;
        }
    }
    if (curBestMatch == null) return(ERROR_NO_COVER);
    N.bestCost = curBestCost;
    N.bestMatch = curBestMatch;
    /*** information has been stored on node N ***/
    return(SUCCESSFUL);
}

/*** sample cost calculation (can be unique for each pattern) ***/
function P.costFunction(faninCosts[]) {
    cost.area = P.area + ∑_i faninCosts[i].area;
    cost.delay =
        /*** note that local latency depends on input ***/
        max_i (P.inputToRootLatency[i] + faninCosts[i].delay);
    return(cost);
}
```

Figure 7.5: Basic tree covering algorithm

complexity, the execution time of the tree covering algorithm is *O(NR)*, where *N* is the number of nodes in the graph (after any duplication from the DAG splitting), and *R* is the number of patterns in the library.

Techniques used to reduce the size of the representation of the pattern library, thereby increasing mapping efficiency, will be described in Subsection 7.5.6.

7.5.3 Placement by tree covering

Since the modules will form a bit-slice datapath layout, a linear ordering of the modules in the datapath must be determined at some point. This placement ordering could be determined in a separate step, following the module mapping. But with this approach, the module mapping step would not have the benefit of knowing whether a given input comes from a module that is far, close, or adjacent, and so it is not known when a specific input must be buffered. Thus delay estimates along all paths are inexact when deciding which inputs to favor in the grouping of nodes. Placement considerations also impact the consumption of CLB resources, particularly input buffering resources, in ways that affect how much computation can be mapped to a CLB/module. For example, if the D register is used to buffer an input, then it cannot be used to receive a load. In the absence of placement information, either optimistic or conservative mappings must be made, with a subsequent peephole optimization pass either to correct illegal over-subscribed mappings or to improve under-subscribed mappings. Either approach leads to suboptimal results.

To address these problems arising from separate module mapping and placement phases, GAMA instead performs them simultaneously. As the covering proceeds from bottom up, the cover of the subtree rooted at each node specifies not only the grouping into modules but also the relative placement of those modules. Thus mapping has exact information regarding distances between modules.

Module layout construction follows the usual approach of dynamic programming: each new candidate solution is formed by combining the best solutions of the subproblems. Specifically, a new layout is always formed by placing the new module at the bottom, then placing the layouts for the fan-in trees above it in some order. The only degree of freedom is the ordering of the fan-in tree

layouts. Note, however, that unlike many dynamic programming algorithms that combine optimal sub-solutions in a way that preserves optimality, this layout approach combines sub-solutions that in no way guarantees the optimal solution.

The resulting layout can be called a “postfix placement” since it corresponds to some postfix traversal of the module tree, with the freedom of visiting children in any order.

The key idea for seamlessly integrating placement determination into tree covering is that each module in the library is replaced with multiple copies, each specifying a different relative placment of its fan-in trees (**Figure 7.6**). With this GAMA extension, the best cover of a tree also specifies the linear order of all modules in the cover. That is because each (sub)tree root specifies the relative placement of its constituent subtrees, and so on recursively to the leaves. When GAMA is trying each potential module pattern, it is simultaneously evaluating not just alternative node groupings but also alternative layouts.

When a pattern and its implied layout is evaluated, exact inter-module distances are deduced by combining the following information: (i) the root module’s specified fan-in ordering (ii) the fact that the output of every subtree layout is available at its bottom edge, and (iii) the known area of each subtree layout. The distance from a non-adjacent fan-in is simply the sum of the rows in intervening subtree layouts.

Because exact inter-module distances are known, there is no need for either conservative or optimistic guessing. Both optimistic and conservative versions of modules are in the library. Optimistic modules need constraints on the distance from inputs—specifically when an input needs to be adjacent or close. If an optimistic module meets all input constraints, it can be used, and will be used when it is cheapest; otherwise a more conservative module is used. An ‘adjacent’ constraint need not be explicitly checked since it follows immediately from which input is placed closest to the root module. Similarly, a ‘close’ constraint need not be checked when the fan-in is the one placed closest to the root module. For a non-adjacent fan-in, a ‘close’ constraint requires a check of the actual distance; if too far, the module/pattern is not a match and is indicated as so by returning infinite cost. The pseudocode for placement-sensitive cost evaluation is shown in **Figure 7.7**.

After the final best cover has been found, the ‘adjacent’ and ‘close’ constraints are added

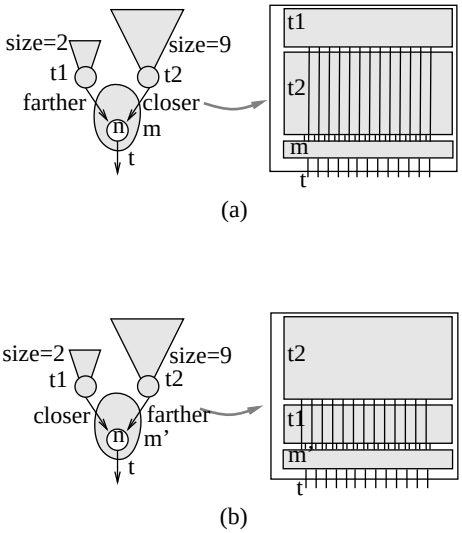


Figure 7.6: Two alternative coverings of node n by two modules, m and m′, that differ only by their fan-in ordering. The different layouts that result are shown to the right. Note that routing lengths and thus delays are different.

to the inter-module edges. This is true even for constraints that did not have to be checked explicitly during covering. A simple example is given in **Figure 7.8**. It shows four modules identical in grouping but differing in fan-in ordering and/or constraints.

Obviously the layout policy used during covering is very restrictive. Likely the real best layout will intermingle the modules from different subtrees. This limitation is compensated to a large degree by global rearrangement of modules after the initial covering-based mapping and placement. In general, the tree covering algorithm for grouping and relative placement mainly helps the critical path. Subsequent global rearrangement and peephole optimization steps then help tidy up areas off of the critical path, improving delay, routing locality, and sometimes area.

7.5.4 Costs

The dynamic programming algorithm only remembers the “best” covering for each partial match at each node in the input graph. In the instruction selection task for which `lburg` was designed, the only cost that mattered was cycle count, whereas in this hardware mapping problem

```
function placementCostFunction(P, fanInCosts[]) {
    cost.area = P.area + ∑_i fanInCosts[i].area;
    for i {
        dist[i] = sum of sizes of fan-in trees
                  placed between fan-in[i] and P,
                  according to P's fan-in ordering;
        /*** throw out matches not obeying placement restrictions ***/
        if (P.mustBeClose[i] and dist[i] > closeThreshold) return(∞);
    }
    /*** P.inputToRootLatency[] factors in any buffering delays ***/
    cost.delay = max_i (P.inputToRootLatency[i] + fanInCosts[i].delay);
    return(cost);
}
```

Figure 7.7: Placement-aware cost function used for evaluating the cover and layout resulting from matching pattern P.

there are two important metrics: delay (critical path delay from any primary input to that node) and size (number of rows of CLBs utilized). Even when optimizing just critical path delay, the sizes of the coverings of subtrees must be known in order to calculate routing delays and make appropriate placement decisions. Thus, the required cost information is actually an aggregate of multiple scalar values. To support this, `lburg` was extended so that the costs are represented by a user-defined structure (where the 'user' here is the person coding GAMA). DAGON also used both area and delay costs. The tree matcher generator tool that it used, Twig [Tji86], already had support for aggregate costs and so did not need modification.

Associated with each pattern in the library is a small C code fragment, supplied by the user, to calculate the cost of the resulting cover if that pattern is matched. The costs of the covers at the fan-ins to the pattern are supplied for use in the calculation. Typically the code fragment is just a call to a subroutine such as costFunction() in **Figure 7.5** or placementCostFunction() in **Figure 7.7**, but it can be customized arbitrarily for each pattern. A pointer to the node itself is also available, so that any information stored on the node (or nearby nodes) can also be used in the cost calculation. For example, the cost calculation for a constant multiplication finds the actual value of

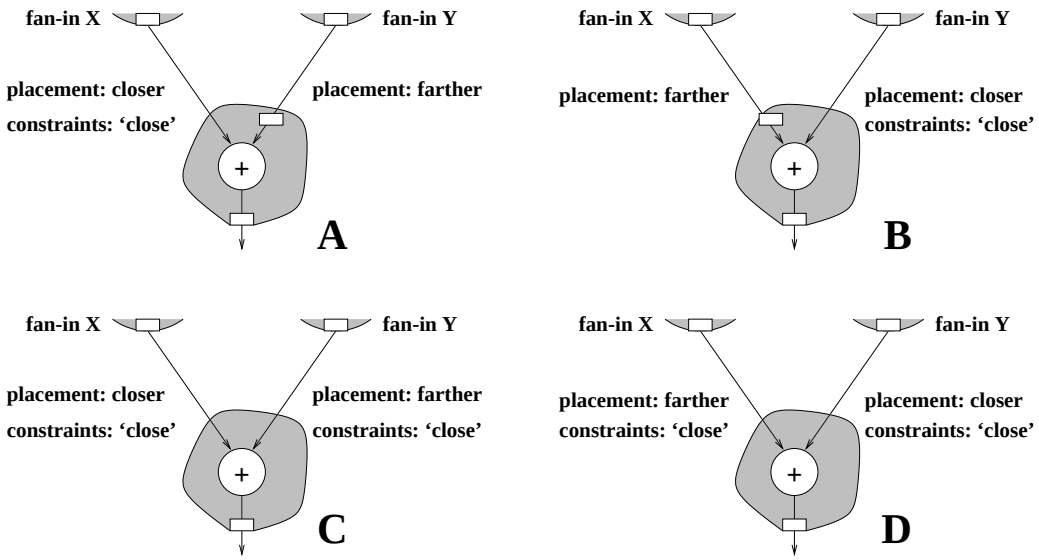


Figure 7.8: Example of modules with same grouping but different in placement ordering and constraints. Because the addition uses the carry chain, inputs must be ‘close’ or buffered locally. Modules A and B are the conservative versions and will match in any situation, although an extra cycle of latency is added to the path from the far fan-in. Modules C and D are more aggressive, assuming both inputs are close. Consider one specific case where an addition node has fan-in X with best cover [delay:6, area:10] and fan-in Y with best cover [delay:6, area:5]. Modules A and B will both match with delay 8 since one of the inputs in each case has a total latency of 2. Module C will not match because with fan-in X adjacent, fan-in Y is far and thus does not meet the required ‘close’ constraint. Module D does meet all constraints and results in a delay of 7, so among these four modules, it would be chosen. However in a real case, it is likely that a module that groups more nodes would be the best match at the addition node unless the addition node is a leaf of the tree.

the multiplication node's constant input and uses it to determine how many rows will be required to implement the multiplier.

There must be a way of comparing the costs in order to determine which candidate is "best". In the basic `lburg`, the lesser of the two scalar costs is selected. In order to handle costs that are arbitrary structures, `lburg` was extended to allow a user-defined macro `BETTER()` that takes two cost structures as arguments and returns true if and only if the first cost argument is better than the second.

There are currently two versions of `BETTER()` implemented. The "area" version favors the cost with smaller area, with delay used as a secondary key in the case of identical sizes. The "delay" version favors the cost with less delay, with size used as the tie breaker. In its basic mode, GAMA uses the same version of `BETTER()` when covering the entire graph. While minimizing just area or just delay is straightforward, minimizing both simultaneously or trading off between the two is not.

7.5.5 Size-delay tradeoffs

The tree covering algorithm is optimal if the goal is minimum area. Picking the smallest solutions to the subproblems will always lead to the smallest solution for the entire tree.

Trying to optimize area and delay simultaneously is not straightforward. In this case it is impossible to pick the single best solution for each subproblem without some global information. For example, if a node is on the critical path, the best cover is probably the fastest one, but if a node is off of the critical path, the best cover is probably the smallest one. But at the time a node is being covered, it is not known whether or not it is on the critical path.

The approach used by GAMA is to first cover the entire tree to minimize delay. This gives an estimate of the ASAP (*as soon as possible*) delay value at each node. The ASAP values in turn can be used to estimate the operation delay at each node. The time constraint at the output of the tree (which must be greater than or equal to the ASAP value at the root node) is then used to calculate the ALAP (*as late as possible*) value at each node. This ALAP value represents a target delay. If it is exceeded at a node, the delay of the overall circuit will not meet the specified timing constraint.

After this estimation, the tree is covered once again, as usual, from leaves to root. As each node is reached for covering, the first attempt at calculating a cover uses the area minimization version of `BETTER()`. If, however, the ASAP value for this cover is found to exceed the previously calculated ALAP goal value (i.e. , all of the slack has been used up), the node is covered again, but using the delay minimization version of `BETTER()`. Thus, along a non-critical path, nodes are covered to minimize area until all of the slack is used up; then for the rest of the path, delay must be minimized. This method is not guaranteed to result in the smallest global solution since the slack is absorbed greedily by nodes nearest the leaves; it could be that a greater area reduction would result if some of the slack were used by a node nearer the root that had a slower but much smaller alternative mapping.

The idea of optimizing area on non-critical paths was also used in Chortle [Fra92], although the details of the implementation are slightly different.

7.5.6 Large module library

As mentioned earlier, there are an extremely large number of different compound modules that can be implemented by a row/column of LUT-based CLBs. This number is further increased by the introduction of placement variations for each module. GAMA uses two techniques to keep the size of the representation of the library manageable.

First, there are many DFG node types that, while computationally different, are equivalent in regards to mapping—they can be packed in exactly the same way into compound modules. When performing the mapping, there is no need to differentiate those node types (opcodes) that are in the same *mapping equivalence class*. The nodes in the DFG as well as those in the patterns in the module library are therefore named by their mapping equivalence class rather than their opcode (**Figure 7.9**). The actual opcode is stored on each DFG node but is not used until module generation.

The second technique to reduce the size of the library representation is factoring out common subpatterns. This technique is commonplace, but is reemphasized here because it is even more important with the large library of patterns with which GAMA must contend.

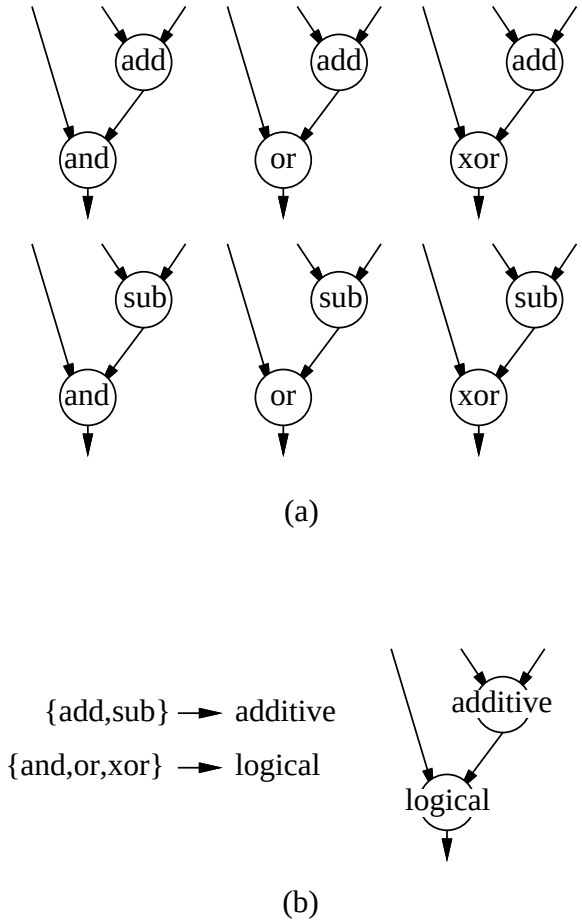


Figure 7.9: Example of library representation size reduction via equivalence. In (a), not exploiting mapping equivalence, there must be a pattern for each combination of opcodes. In (b), the equivalences between addition and subtraction and among bitwise-logical operations are exploited so that this single pattern can replace the six patterns above.

7.6 Scheduling and execution (non-pipelined)

Some modules such as those involved in Exits and memory accesses must be triggered only during the appropriate cycle of each iteration, since execution before their inputs are valid would have undesired consequences. Also, the Hold modules used with non-pipelined execution need a trigger at the end of each iteration so that they load the loop-carried values for use during the next iteration. These triggers are provided by a simple *sequencer* that is synthesized within the configuration. Its duty is to keep track of the current cycle within the iteration and provide triggers to modules as needed. The implementation of the sequencer is trivial since every iteration has the same fixed schedule of SL cycles where SL is the schedule length. Conceptually it is simply a Boolean shift register of SL stages, hooked back on itself, passing around a single '1' bit. The actual implementation is slightly more complicated; it may in fact have side chains forking off of the main chain to allow for more localized routing.

Modules such as simple adders need no trigger from the sequencer as they can execute constantly with no ill side effects. However, it is useful to think of these non-triggered modules as also being scheduled for a specific cycle: the cycle of the iteration that their output first becomes valid.

In most cases a module is "scheduled" exactly at its ASAP (as soon as possible) value. This is equal to the greatest total latency along any path (composed of any combination of distance-0 data, precedence, and/or liveness edges) from a Hold module to that module's output. The one exception occurs when there are multiple non-queue memory accesses. Since each such access requires exclusive use of the address bus, they cannot be scheduled for the same cycle. If two accesses have the same ASAP value, the one with the least slack is given preference and gets scheduled in that slot, while the other is delayed to the next available cycle; consumers of the delayed access are also then delayed from their ASAP value. Non-queue memory accesses present the only case where per-cycle resource constraints must be considered.

With non-pipelined execution, a module's output is guaranteed to be the correct value for that iteration from its scheduled cycle until the end of the iteration. Outside of that range, the

module's output must be assumed to be incorrect for either the prior or current iteration, since in general the module's inputs may be a combination of values from the prior and current iteration. This hints at a key idea for pipelined execution (Chapter 8), with which a module's inputs always arrive synchronized for a particular iteration.

7.7 Limitations

There are several reasons why the mapping and placement solution given by GAMA is not optimal. The initial splitting of the input DAG into trees means that nodes in different trees cannot be combined into a single compound module, which could prevent the optimal solution for the DAG from being found.

As described earlier, when the optimization goal considers both area and delay, GAMA cannot guarantee finding the optimal solution when only the single best solution is kept at each node. This could be improved by keeping instead the set of solutions forming the best curve of tradeoffs between area and delay (discarding any clearly subtoptimal solutions). When evaluating a module cover, all combinations from the fan-in covers' sets would have to be evaluated to form the new best curve of solutions.

The restricted module placement forcing a module subtree's output to always be at the bottom edge of its layout similarly limits the number of potential solutions considered, likely excluding the best one from consideration. This could also be improved by keeping multiple solutions at each node, with different solutions having the output at different locations within the layout. This could allow bidirectional assembly of datapaths. Perhaps all variations could be kept—with the output in row 1, row 2, etc. upt to the last row—or just a subset—first, last, and middle—could be kept.

Finally, since GAMA can only choose from modules in the library, the solution can only be as good as the module library. This is considered in the next subsection.

7.8 Comparing module library to instruction set

The final datapath can be viewed as a group of static, complex instructions interconnected by buses. This inspired the derivation of the module mapping algorithm from the instruction selection algorithm. These both highlight the similarity between designing the module library and designing an instruction set. This section compares and contrasts in factors in designing "set of instructions in instruction set architecture (ISA)" with "set of modules the generator knows how to build".

Both module set and instruction set design hope to identify groups of operations that can be implemented as an optimized unit to give better performance. However, differences in how the groups are implemented make a great difference regarding when it makes sense to add a new grouping.

Instruction set designers must weigh negative aspects of each potential new addition. Every new instruction impacts the microprocessor's implementation even when it is not used. The control/decode logic and possibly the datapath must be modified to support each new instruction. This in turn affects design complexity and perhaps performance.

The module library, in contrast, only comes into play during compile time, and thus adding a new module has no negative impact on run-time performance. Essentially the module library attempts to capture a good designer's expertise regarding how datapath operations can be packed into a small module, and more knowledge is better. The main drawback to a large module library instead is increased complexity, making it more difficult to maintain the library and associated module generator. Also, the speed of GAMA in performing the tree covering is also impacted by increased size of the module library; however, GAMA mapping is so fast compared to other compilation phases that it is very unlikely that this consideration would become significant.

With instruction set design, the total number of different instructions is bounded by their encoding into typically 32-bit instructions including immediate data values and register. Thus inclusion of one opcode may preclude the inclusion of another. There is no such encoding limitation with the module library.

Another related difference is that module library design does not have the legacy problem. Superseded modules can be removed from the library at any time with no ill effects; datapaths synthesized using the old module library still execute perfectly. With instruction set design, in contrast, compatibility requirements mean that an ill-chosen instruction must be supported over lifetime of that and any derived ISAs, complicating future generations of implementations.

Chapter 8

Pipelined Execution

Although predicated and speculative execution help expose operation level parallelism within an iteration, much greater amounts of parallelism can be exploited, and thus greater throughput achieved, by looking across iteration boundaries for operations to perform in parallel.

The simplest way to schedule operations across multiple iterations is to perform loop unrolling prior to scheduling. Loop unrolling concatenates N copies of the original loop body to create a new larger loop body, taking care that the new loop body behaves the same as the original when N does not exactly divide the actual number of iterations. Loop unrolling allows a normal (acyclic) scheduler to schedule together the operations from N copies of the loop bodies. Still, every N (original) iterations there is a scheduling barrier.

Loop unrolling has an obvious drawback with fully spatial computing: it increases the amount of reconfigurable resources needed for computation by roughly a factor of N. The decision of when to apply unrolling, and then by what factor, would need to be integrated with pruning since both involve area-performance tradeoffs. For example, applying an additional prune might result in more iterations exiting the kernel, but would allow the kernel to be unrolled by a factor of two—would it be worth it? Unrolling would also be risky combined with inexact estimates of area; the unrolled kernel may turn out to be too big after synthesis, requiring the loop to revert to software implementation.

Software pipelining is a different compiler technique for scheduling operations across

iteration boundaries. It involves no replication of computation resources within the loop and can often directly achieve the optimal increase in parallelism. It will become clear and is likely obvious to many readers that this approach is usually superior to unrolling with spatial computing. This is the approach used by `garpcc`.

Although software pipelining can be useful even with single-issue processors, it is usually associated with VLIW processors that can exploit larger amounts of operation level parallelism under the guidance of the compiler. Indeed, most VLIW architectures include specific features to efficiently support software pipelining. A survey of software pipelining techniques can be found in [AJLA95]. The approach used here is inspired mainly by Rau's work at Cydron and then at HP-Labs with Schlansker [RG81, Rau96, RST92], as well as further work by the Impact Compiler group at the University of Illinois where pipelined scheduling was performed on superblocks/hyperblocks [LH96].

There is some irony in the fact that `garpcc` derives its pipelining approach from software pipelining, since software pipelining was derived in turn from pipelining in hardware circuits, which at first glance more closely resembles spatial pipelining on the Garp array. Indeed, much of the compiler and architecture complexity introduced to efficiently support software pipelining on VLIW processors is not required when using the Garp array, since pipelining is naturally a spatial technique. Yet there are two key aspects in which the pipelining problem for `garpcc` resembles software pipelining: modules are scheduled on fixed clock boundaries; and scheduling must avoid global resource conflicts between modules scheduled for the same cycle. Both these considerations are shared with software pipelining while neither is considered with hardware "retiming" approaches that locally rearrange logic and registers attempting to increase the clock rate. "Software pipelining" is also consistent with previous techniques and related terminology borrowed from VLIW compilation.

Combining both unrolling and pipelining can lead to additional optimization opportunities as described by Lavery [LH95], but this has not been investigated in `garpcc`.

8.1 Minimum initiation interval

Without pipelining, an iteration's execution begins only when the previous one's has finished. Thus one iteration is started every SL cycles where SL is the number of cycles in the schedule of module execution. This usually equal to the number of cycles in the critical (longest) path through the datapath modules within an iteration—from one Hold node to another. The only exception is when competition for the memory port requires extension of the schedule length.

With pipelined execution, an iteration starts before the previous finishes, following the earlier by II cycles using the same modules where II is the initiation interval and is less than or equal to SL. Two effects limit how closely one iteration can follow another, acting as lower bounds on the II. The two effects are the same as with VLIW software pipelining [Rau94]:

Recurrence-limited minimum initiation interval (RecMII) A recurrence—a cycle of edges of any mixture of types in the DFG (pre-mapping estimation) or module graph (post-mapping)—limit how soon an operation in one iteration can execute after the same operation of a previous iteration. Since all iterations have the same schedule, this directly affects the II. Considering a DFG cycle with cumulative latency of L clock cycles and crossing D iteration boundaries, an operation in iteration $i + D$ occurs $D \times II$ clock cycles after the same operation in iteration i. Those two occurrences cannot occur less than L clock cycles apart, giving the constraint $D \times II \geq L$, or rearranging and considering that a non-integer II is not possible without unrolling, $II \geq \lceil L/D \rceil$. The recurrence-limited lowerbound RecMII is given by the largest such constraint considering all cycles. Any cycle equaling the largest constraint is called a critical cycle.

Resource-limited minimum initiation interval (ResMII) Competition for the memory port can also limit how closely one iteration can follow another since only one non-queue memory access can be initiated each clock cycle. If each iteration performs N non-queue memory accesses, starting each new iteration with an interval of less than N is not sustainable. Thus the resource-constrained lower bound on II, ResMII, is equal to the number N of non-queue memory accesses (including those queue-legal accesses in excess of the three available queues). In comparison, the computation

of ResMII for VLIW processors is much more complicated since in general there are overlapping categories of resources—issue slots, function units, memory ports, etc.— that must be considered.

Combining these two, the minimum initiation interval MII is the maximum of these two lower bounds. MII may not be achievable, for example when there are two cojoined critical cycles having a mutual resource conflict. Even when an initiation interval II = MII is theoretically achievable, it may not be achieved by the scheduling algorithm presented later.

8.2 Changes to other compilation phases

In `garpcc` pipelining is accomplished mainly through rescheduling the execution of the modules on the array, which is one of the last compilation steps. However, to get the most out of pipelining, two changes are needed during earlier phases of compilation. Both result from the fact that pipelined performance is governed by the initiation interval—usually determined by the critical cycle—and not the schedule length.

- When considering a path to include or reject from the loop, the pruning algorithm now evaluates the impact on MII rather than on the schedule length.
- The module mapping algorithm, when packing DFG nodes into feasible modules, adjusts its goal to minimizing the estimated critical cycle instead of minimizing the critical path.

After module mapping, each module is considered as an atomic unit. Its scheduling in time as well as its placement on the array are still flexible. Recall that modules are completely pipelined, with a register inserted every cycle of delay on internal paths. Data, precedence, and liveness edges are all retained in the graph of mapped modules.

8.3 Modulo scheduling algorithm

Because multiple iterations will be active simultaneously on the network of modules, care must be taken so that non-queue memory accesses from different iterations do not conflict. `Garpcc` utilizes a scheduling algorithm directly based on Rau's iterative modulo scheduling (IMS) [Rau94, Rau96].

Modulo scheduling is a framework for scheduling a single iteration of a loop in a way that resolves resource conflicts among consecutive overlapping iterations [RG81]. Successive iterations have identical schedules, shifted by a number of cycles called the initiation interval (II). Resource conflicts between overlapping iterations are captured by folding the schedule of the single iteration back on itself, modulo II cycles.

In brief, `garpcc` implements IMS as follows. A lower bound on II is found by using the greater of the recurrence-constrained minimum II and the resource-constrained minimum II. IMS attempts to find a valid schedule for this minimum II; if it fails, it tries again with an II incremented by one, and repeats until a valid schedule can be found. A worklist algorithm is used to search for a valid schedule for each given II. The worklist contains unscheduled modules, initially containing all modules. As each module is removed from the worklist, it is scheduled for the earliest cycle in which it does not conflict with any other scheduled module in terms of precedence or modulo resource constraints. If no such cycle exists, the module is still scheduled for some cycle, but other conflicting scheduled modules are descheduled and returned to the worklist. The algorithm terminates successfully if the worklist becomes empty (all modules have been scheduled). It terminates with failure for that value of II if *Budget* scheduling steps have been performed and there are still unscheduled modules. *Budget* is set to a small multiple of the number of modules.

After modulo scheduling, the compiler removes extra slack on edges between different strongly-connected components (SCCs) where possible. If an SCC has no global resource usage, it can be shifted arbitrarily while still leaving a valid schedule. If it does use global resources, it can be shifted only by multiples of II cycles.

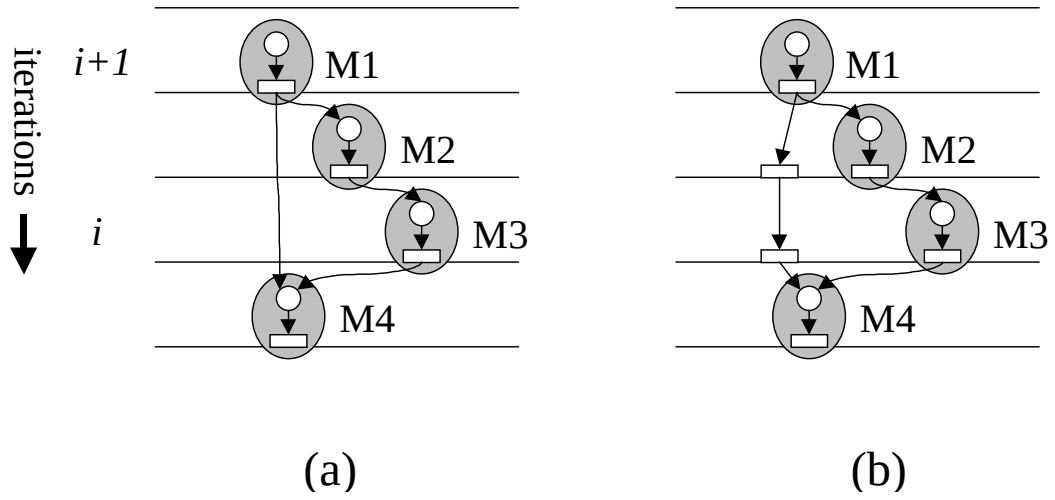


Figure 8.1: Register insertion for pipelined execution with II=2 (see text). (a) M4 will incorrectly see operands arriving from different iterations. (b) After register insertion, operands arrive at M4 correctly.

8.4 Register insertion

The overlap of iteration execution causes a problem as shown in Figure **8.1**(a). Consider a schedule with II=2. Module M1 will produce its output for a certain iteration i at cycle 1 of that iteration. That value is correctly consumed immediately by module M2. However, a problem develops with module M4 at cycle 4 of iteration i. At that point, module M1 has already computed its output for the following iteration $i + 1$, but M4 is expecting to receive M1's output for iteration i (although at the same instant, M2 is in fact expecting M1's output for iteration $i + 1$). This problem directly corresponds to the overlapping lifetime problem encountered in VLIW compilation.

The solution is to insert registers on edges such as that from M1 to M4 (Figure **8.1**(b)). This ensures that partial results from the same iteration arrive at a module's inputs with the correct timing. The progress of an iteration—its partial results and external actions such as memory accesses—is confined to exactly one level in the schedule. This ensures that multiple iterations can utilize the hardware without interfering with each other.

Each data and liveness edge is examined to see if it requires the insertion of registers.

The insertion of registers on liveness edges ensures that the correct version of a variable is available when an exit is taken, and is somewhat analogous to the modulo variable expansion to remove live-out anti-dependences described in [LH96].

The number of registers that must be added to an edge is equal to the slack on the edge after scheduling; this is why the compiler attempted to remove excess slack between SCCs after modulo scheduling. Edges in critical cycles will not need any additional registers. If a module has fanout, the same register chain is shared by the different consuming modules.

The slack on an edge can be calculated after the schedule of the modules has been fixed. Where $\sigma(n)$ is the scheduled cycle for module n, and $lat(E)$ is the minimum scheduling difference considering edge E only, the slack on edge E is given by:

$$slack(E) = \sigma(sink(E)) - lat(E) - \sigma(source(E))$$

This formula can be generalized to handle inter-loop as well as intra-loop edges as follows:

$$slack(E) = \sigma(sink(E)) + II \times dist(E) - lat(E) - \sigma(source(E))$$

In this formulation, Hold nodes for loop-carried variables have been removed from the DFG; for each Hold node removed from along a data edge, the edge's distance is incremented by one.

Since timing on all paths is now exactly synchronized, there is no need for the Hold modules on each loop carried edge. Just as data is shifted synchronously forward, the passing of loop-carried variable values is timed to exactly meet the values of the correct subsequent iteration. All cycles in the module graph will contain exactly $D \times II$ clock cycles of delay, where D is the total number of iteration boundaries crossed by the path: $\sum_{E \in cycle} dist(E)$.

After registers are inserted, placement is again performed on the datapath so that the new register-only modules intermingle with the original modules. When one or more registers are inserted along an edge that has a placement constraint ("close" or "adjacent"), that constraint is transferred to the edge between the last inserted register module and the original edge destination. No placement constraints are needed on edges between two inserted register moduless or between the original source and the first register module.

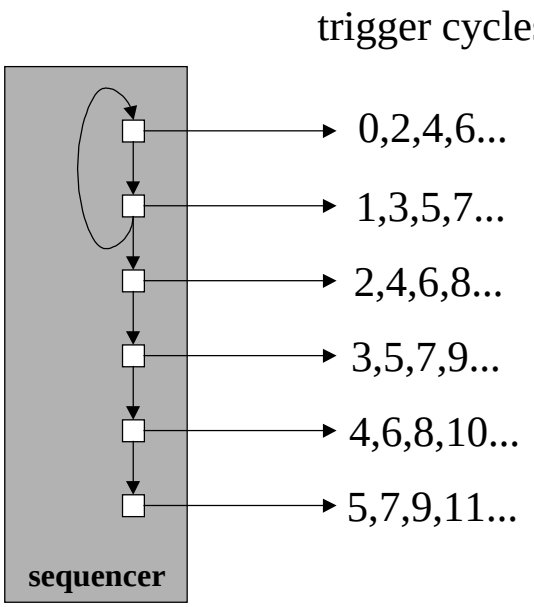


Figure 8.2: Sequencer for pipelined execution with II=2.

Adding registers can be inefficient when II is moderate to large. Considering a long chain of inserted registers, only one out of every II registers actually contains a useful value. The other II-1 out of II registers contain "don't care" values. When II is large it is more efficient to instead use Hold modules. In the case of Garp, it is beneficial to use Hold modules instead of simple registers when II > 2, since each row can implement either one Hold module or two simple registers. These Hold modules are triggered once every II clock cycles, so that the values are shifted once every II clock cycles rather than every clock cycle. The number of Hold modules needed at the output of a module is $\lceil slack(E_M)/II \rceil$, where E_M is the module's output edge with the most slack. Note that in this use, a Hold module does not necessarily correspond with the crossing of an iteration boundary.

8.5 Sequencer changes

The sequencer's implementation was trivial with non-pipelined execution, and is still quite simple with pipelined execution. Although the sequencer's shift register length is still equal to the schedule length SL [1], the feedback loop is different, so that subsequent iterations are started before the current is done. Specifically, the feedback edge originates from the II'th tap rather than the SL'th tap. Thus, a sequencer output from tap t is activated at cycles t, $t + II$, $t + 2 \times II$, etc. (Figure **8.2**).

8.6 Prologue and initial values

Pipelined computation, whether in software or hardware, characteristically has a *prologue*, the period from the start of computation to the time the first iteration completes, at which point the pipeline achieves a steady state. Specifically, during the prologue, latter portions of the pipeline should not be active, since the first iteration has not yet reached them. With VLIW processors, a number of schemas for handling this have been put forward [RST92]. One category emits the prologue as separate code, with inactive pipeline stages simply eliminated. Another solution, termed "kernel-only", uses the same instructions for both the prologue and the steady-state execution, but uses *stage predicates* to suppress inactive stages during the prologue. The sequencing technique utilized by the Garp compiler more closely resembles the latter, since each iteration maintains the same schedule, even those started during the prologue. In some sense, the sequencer combines the functions of program counter (activating "instructions" 0...II-1) with stage predicates.

Loop-constant inputs are handled the same as in the basic schema. Before array execution begins, they are transferred into their appropriate registers where they remain constant throughout all iterations. Similarly, queues are initialized by the main processor in the same fashion as with non-pipelined execution.

Initial values of loop-carried variables are handled differently, however, because Hold modules will not necessarily exist on every loop-carried data edge. `Garpcc` could build special

[1] The schedule length SL may have been extended compared to the non-pipelined case in order to resolve modulo resource conflicts.

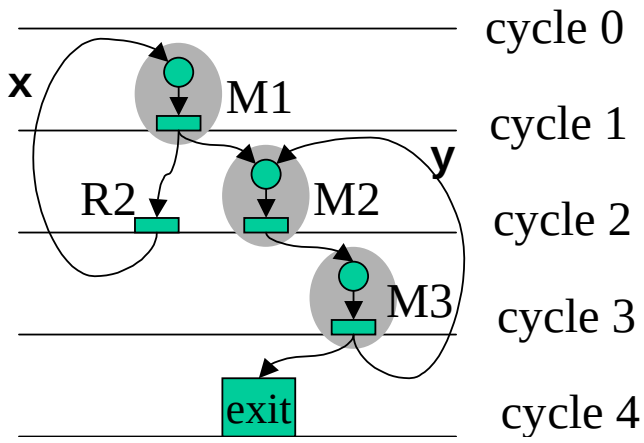


Figure 8.3: Loop-carried variable initialization during prologue. II=2. The microprocessor writes the value of variable 'x' to register R2 at cycle 0 for use by M1 during the first iteration. Similarly, the value of variable 'y' is written to the output register of M3 at cycle 1 for use by M2.

circuitry into the datapath to handle variable initialization, but this would add complexity and possibly slow down the steady-state pipelined execution rate. So instead, `garpcc` utilizes the Garp architecture's close connection between the MIPS processor and reconfigurable coprocessor, and transfer the complexity into software (the software does not perform the prologue, it simply helps provide the initial values during the prologue).

Each module input that is fed by a loop-carried data edge must be supplied an initial value during the first iteration (Figure **8.3**). For each such input, the main processor writes into the supplying register the correct initial value at precisely the correct cycle for use by the first iteration. Since there may be a number of such initial values required at different cycles of the prologue, the processor will typically alternate between writing one or more initial values, then activating the array for a small number of cycles, then writing another initial value, and so on until all initial values for use by the first iteration have been supplied. After the last initial value has been supplied, the processor activates the array for an indefinite period, that is, until the array halts itself.

If a loop-carried variable is used by multiple modules, usually only one write of the initial value is required. If the consuming inputs are all required the same cycle, they would be supplied by the same register (whether the source module's output, or a delayed version thereof). If the

```
do {
    x = x+y;
    y = x;
} while (...);
```

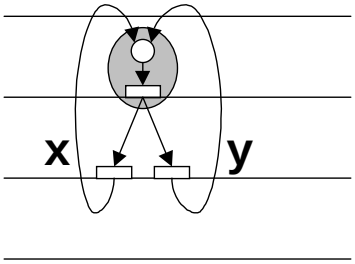


Figure 8.4: Special case for aliased variables.

consuming inputs are needed different cycles, only the earliest-used register in the chain need be initialized; this initial value will then propagate down the rest of the chain supplying later-needed inputs. One way of thinking about this initial value of the variable is that it appears as the output of the module from a fictitious iteration –1.

A rare but important special case occurs when two loop-carried variables are aliases, that is, when one variable is a copy of the other at the end of the iteration (see **Figure 8.4**). The SSA form has removed any difference between uses of the two variables—all consumers will be connected to the module that actually produces the value. This is correct for the loop-carried values into the second iteration, third iteration, etc. The problem is that the two variables will in general have *different* initial values for use in the first iteration. If the two different variables are used during the same cycle, obviously the same register cannot be used to store the two different initial values. This situation is handled by forking the register chain, so that two different registers are available for initial values, but on subsequent iterations the two forks are supplied by the same value. If this aliasing occurs on a critical cycle so that there is no delay chain to fork, a minimal fork of one register for each variable is inserted, and the II is increased by one. An alternative would be to duplicate the module producing the value so that one exists for each variable, providing a separate place for each initial value; this solution has not been implemented due to the rareness of this situation.

8.7 Epilogue

Software pipelining of counted loops traditionally involves the generation of an epilogue: special code to finish off the iterations that are in progress at the point when the last iteration is started.

The Garp compiler takes a simpler approach that does not require an epilogue: it simply allows the last iteration to complete using the array, then discards work that has started speculatively on subsequent iterations. This approach is possible because of the Garp architecture's support for speculative execution, and in particular, speculative loads (a subsequent iteration might execute a load with an invalid address). Besides being simpler, this approach also has the benefit that it works for loops with data-dependent exits, allowing a uniform pipelining schema for all loops.

It could be argued that a special epilogue should be generated for counted loops, since it can be optimized to avoid doing unnecessary work. However, in the case of Garp, in practically all cases an optimized epilogue in software is still much slower than simply completing the last iteration in hardware, even if 'extra' work is done in the latter case.

The copying of registers from the array to the main processor register file at loop exit resembles the "live out copying" required by many VLIW software pipelining schemas [RST92, LH96]. They are similar in that the source of each live variable value, and even *which* variables are live, depends on which exit is taken. However, with Garp this copying must be performed even when reconfigurable hardware execution is not pipelined.

Chapter 9

Experiments

This work set out to implement a complete automatic compilation path from C to the Garp chip, and then measure effectiveness and identify the weak points of the compilation path. The latter work required the compilation and simulation of real programs and analysis of the results.

9.1 Methodology

9.1.1 Garp simulator

No Garp chip exists as real silicon; performance evaluation is carried out via simulation. The direct, full simulation approach is used, i.e. it is neither trace-driven simulation nor instrumented direct execution. Although it is the slowest, the full simulation approach is the most accurate and allows the most flexibility for tracing execution.

The simulator loads a single, real Garp executable exactly as would run on a Garp chip if one existed. No shortcuts were used between the compiler and simulator so that the impact of all compilation and decoding details would be taken into account.

The simulator handles a single process, and thus no context-switching overhead is measured. All library code and parts of operating system (OS) kernel code, including emulation of floating point instructions, are directly simulated. Most system calls are eventually executed on the host. For example, an `fopen()` call from the user's program will execute as follows. All in

library code linked with the executable, `fopen()` calls `_fopen_r()` which calls `_open_r()` which calls `_open()` which finally issues a `syscall` instruction. This transfers control to the OS kernel, still in simulated Garp execution. The OS kernel saves all registers as if in preparation for performing the actual file open, but only then some magic occurs; an `open()` call is executed on the host (the machine on which the simulator is running). Appropriate structures and values such as the file descriptor are recorded for use in mapping state between the simulation and actual open files on the host. When data transfer is involved, as in the case of a `read()` call, translation between big endian and little endian formats may be required depending on the host architecture. The only time not summed into the results is that for performing the call on the host and doing necessary bookkeeping and data translation.

One feature of the Garp simulator is that all experiments are reproducible. The exclusion of any time spent on the host, particularly actual file input/output, helps much. The remaining details are `gettimeofday()` and `time()` calls, typically used for seeding random number generators. The simulator has a command line option causing them to return deterministic values, incremented by a fixed amount each time called.

9.1.2 Garp implementation details

The Garp specification describes a small family of upwardly-compatible architectures. The architectures differ in the number of rows in the array, which fixes the maximum size of a legal configuration. Both `garpcc` and the simulator support all Garp architectures from 32 to 1024 rows via a command-line option. Unless otherwise stated, a 32 row array is assumed in the following experiments.

For any basis of describing performance, details of a particular implementation of an architecture must be assumed by the simulator. Implementation details include factors that need not be known by the compiler to generate correct object code, such as clock rate and cache sizes.

Most details pertaining to memory system are modeled after the Sun Ultrasparc 1/170 and are considered fixed. There are separate on-chip 16kB direct-mapped level-one (L1) instruction and data caches, both with 32-byte lines. The L1 miss penalty is 5 cycles. The L1 data cache employs

a hit-under-miss strategy whereby it can continue supplying hit data while servicing a miss; a stall occurs only when a second miss occurs, or when the first miss was a load and an instruction accesses the register that was the target of the load. There is an 512 kB off-chip level-two (L2) cache, also organized by 32-byte cache lines. The L2 miss penalty is either 13 or 22 cycles depending on whether the DRAM page changes.

Implementation features of the three memory queues are assumed as follows. Each queue has a buffering capacity of 512 bytes regardless of the access size for which they are configured. Depending on its configuration a queue accesses either the L1 or L2 cache a cache line at a time (32 bytes). The queues access memory either when necessary, or when both the main processor is not accessing memory and another bulk transfer is possible (i.e. a full cache line is ready for writing or the queue's buffer has another cache line's worth of space available).

Timing details related to the interfaces between Garp's MIPS processor, reconfigurable coprocessor, and memory system are considered fixed, based on partial VLSI layouts of the important critical paths. They are likely conservative.

The size of the configuration cache is an aspect of the implementation that can be varied in the simulator by a command line option. The default is 4 levels.

One feature of Garp is that the clock frequencies of the MIPS and array can differ, and can differ by different ratios on different implementations. For simplicity, the Garp simulator assumes both clock rates are the same. The clock rate for a 32-row Garp array is estimated to be 133MHz in the same VLSI process as the 167MHz UltraSPARC; this estimate is conservative [Hau00]. The clock rates of versions with greater than 32 rows have not been estimated. Obviously much higher clock rates for both the microprocessor and the Garp array would be possible with contemporary fabrication processes.

9.2 Benchmarks

Only integer benchmarks were studied. The eight applications from the SPECint95 benchmark suite are included. However, the "reference" datasets were not used since they were too large

Category	Benchmark	notes
SPECint95	go	-no-forward-prop
	m88ksim	
	cc1 (gcc)	-no-forward-prop -no-predep -no-pointer
	compress	
	li	-no-pointer
	ijpeg	-no-pointer
	perl	-no-pointer
	vortex	-no-pointer
Multimedia	wavelet image compres	-no-forward-prop
	mpeg2decode	
	pegwit	-no-forward-prop
Miscellaneous	gzip	-no-pointer
	cpp	-no-pointer

Table 9.1: Table

to be practical given the simulation slowdown. In some cases the “test” dataset was used; in other cases a subset of the “reference” dataset was used. The datasets were sized to give execution times of approximately one second on a 200MHz Ultrasparc. Multimedia benchmarks come mainly from the Berkeley Multimedia Workload [SS00]. A wavelet image compressor is included from the Stressmark benchmark suite [KPP+98]. Finally, some miscellaneous character-based programs were also included: GNU gzip and GNU cpp. The benchmarks are listed in Table 9.1.

Two of the SPECint95 programs did in fact perform some floating point operations. Since the simulated Garp implementation has no floating point unit, all floating point instructions cause an “unimplemented instruction” trap and are emulated by the OS kernel. Floating point emulation requires several hundred cycles per invocation, for example, approximately 1000 cycles for a multiplication and 2500 cycles for a division. Thus a benchmark with even a small frequency of floating point operations can spend a large percentage of execution time in the OS kernel emulation routines.

The main case was `perl`. Even though the input scripts do not cause any floating point computation to be performed, the `eval()` procedure, which calls itself recursively, contains a local floating point variable that is saved to the stack during calls; these saves involve FP register-memory instructions that are emulated by the OS kernel.

Another case was `compress`, but in this case the floating point operations were eliminated. The floating point operations occurred in the test harness used to generate the input dataset. The modified benchmark instead loads from a file the input dataset generated by the original harness routine.

Some programs could not be compiled with all optimizations. In those cases, the trouble-causing optimizations were turned off. Most commonly the pointer analysis utilized did not complete within reasonable time and had to be deactivated. The option '`-no-predep`' inhibits FOR loop normalization which attempts to set the loop step to unity, allowing dependence analysis to be more effective. It has a bug which caused problems in only some benchmarks. The option '`-no-forward-prop`' inhibits the SUIF pass '`porky -forward-prop`', which caused problems with some benchmarks as well. Such cases are noted in the table.

9.3 Results

9.3.1 Speedup and HW/SW breakdown

The bottom line is whether `garpcc` can effectively utilize the Garp array to speed up applications considering all effects. Table 9.2 compares the `garpcc` results to pure `gcc` compilation as well as the SUIF-to-`gcc` software path that is shares the same handicaps as `garpcc`. `Garpcc` and SUIF-to-`gcc` both suffer relative to pure `gcc` compilation because of the divorce between front-end and final compilation; both convert the SUIF representation back to C for final `gcc` compilation. They suffer because `gcc`'s backend optimizations do not have all of the original high level information even though both paths try to reconstruct control-flow structures before emitting the final C code from SUIF. A particular weakness is SWITCH/CASE statements, which are decomposed to a sequence of (two-way) conditional branches. The original SWITCH statements cannot be reconstructed and so `gcc` cannot generate efficient jump table machine code.

Table 9.2 also provides auxiliary information regarding the percentage of hardware utilization as well as the overhead consumed for utilizing the Garp array. "% HW Compute" time includes both prologue and steady-state pipelined execution even though some overhead work mov-

Benchmark	Speedup vs. GCC	Speedup vs. SUIF	% HW Compute	% HW Overhead	# Kernels	# Kernel Executions
go	0.82	0.84	5.44%	15.06%	96	525491
m88ksim	1.05	1.06	18.15%	1.52%	15	152221
gcc	0.80	0.99	2.10%	2.17%	147	112796
compress	0.98	1.04	17.53%	7.48%	5	654763
li	0.94	1.01	0.01%	0.00%	3	185
ijpeg	1.03	1.09	6.40%	3.39%	42	333691
perl	0.82	1.01	0.55%	0.18%	8	10659
vortex	0.94	0.99	0.16%	0.17%	20	10644
wavelet-image	2.60	2.80	66.74%	14.43%	11	8892
mpeg2decode	0.97	1.04	8.02%	0.88%	18	129908
pegwit	1.04	1.07	4.65%	0.94%	11	46498
gzip	1.19	1.51	38.87%	9.38%	15	30110
cpp	1.14	1.25	7.75%	9.81%	35	41128

Table 9.2: Benchmark execution on Garp.

ing operands to the array may be done during the prologue. "% HW Overhead" includes reconfiguration, moving operands to the array before the prologue starts, exit determination when there are multiple kernel exits, retrieving values from the array, and queue setup and flush overhead.

The hardware execution times are compressed in cases where the kernels provide considerable overall speedup.

The numbers are modest both in fraction of computation using the array and net speedup of the application. One clue regarding the net performance comes from the large amount of array overhead time for some benchmarks. The kernel selection heuristic optimistically estimated that each configuration would be loaded just once. The next subsection investigates how close this matches reality for these benchmarks.

9.3.2 Configuration cache miss rate

When making decisions, the `garpcc` compiler optimistically assumes that the configuration cache is effective, so that each configuration is loaded just once over the entire program execution. If this assumption is for the large part false, then `garpcc` likely makes many bad deci-

Benchmark	Miss rate percentage							
	1	2	4	8	16	64	128	perfect
go	60.728	38.448	19.176	7.457	2.957	0.020	0.018	0.018
m88ksim	73.447	4.032	0.011	0.010	0.010	0.010	0.010	0.010
gcc	37.263	17.262	4.431	1.503	1.301	0.604	0.130	0.130
compress	0.002	0.001	0.001	0.001	0.001	0.001	0.001	0.001
li	2.703	1.622	1.622	1.622	1.622	1.622	1.622	1.622
ijpeg	2.433	1.144	0.953	0.890	0.331	0.013	0.013	0.013
perl	60.784	1.379	0.159	0.150	0.150	0.150	0.150	0.150
vortex	29.979	7.788	0.658	0.188	0.188	0.188	0.188	0.188
wavelet-image	20.434	0.247	0.135	0.124	0.124	0.124	0.124	0.124
mpeg2decode	16.562	3.623	1.049	0.023	0.014	0.014	0.014	0.014
pegwit	29.840	27.203	9.252	0.047	0.047	0.047	0.047	0.047
gzip	72.089	24.268	0.063	0.053	0.053	0.053	0.053	0.053
cpp	42.689	29.911	12.082	3.351	0.246	0.085	0.085	0.085

Table 9.3: Configuration miss rate percentages for different configuration cache sizes

sions. Results summarizing configuration cache miss rates are presented in Table 9.3, both for the standard configuration cache size (four planes) and for larger and smaller configuration caches. The last column shows the miss rate for an ideal cache, where the only misses are compulsory misses (the first time a configuration is accessed).

Starting from the right of the table, as is expected, the miss rate approaches perfect while the number of levels of configuration cache exceeds or is close to the number of configurations. However, it is striking how quickly the miss rate increases for some benchmarks going to the left (towards smaller cache) past that point. Each application varies in the size of its typical "working set" of configurations. '`m88ksim`' does well with four levels of cache, while the curve elbow for '`vortex`' is at eight levels. The benchmarks with large numbers of configurations have no clear-cut working set; they continually improve the more cache is provided. In particular, '`go`' has a cache miss on nearly a fifth of uses of the reconfigurable array with the default 4 levels of configuration cache. Further analysis showed that, even worse, in many cases many of the cache lines holding the missed configuration also missed in the L2 cache, adding yet more latency.

Obviously the compiler could actively reduce the number of kernels to relieve competition

Benchmark	# Kernels	% HW Compute	% HW Overhead	Config cache miss rate %	Performance v SUIF
go	45 v 96	1.83 v 5.44	4.11 v 15.06	9.16 v 19.18	0.97 v 0.84
m88ksim	10 v 15	18.08 v 18.15	1.51 v 1.52	0.01 v 0.01	1.07 v 1.06
gcc	100 v 147	1.47 v 2.10	1.19 v 2.17	2.40 v 4.43	1.00 v 0.99
compress	3 v 5	12.20 v 17.53	5.07 v 7.48	0.00 v 0.00	1.04 v 1.04
li	1 v 3	0.00 v 0.01	0.00 v 0.00	20.00 v 1.62	1.02 v 1.01
ijpeg	23 v 42	6.01 v 6.40	2.16 v 3.39	0.80 v 0.95	1.09 v 1.09
perl	5 v 8	0.48 v 0.55	0.14 v 0.18	0.12 v 0.16	1.00 v 1.01
vortex	8 v 20	0.03 v 0.16	0.08 v 0.17	0.23 v 0.66	0.99 v 0.99
wavelet-image	9 v 11	60.11 v 66.74	11.93 v 14.43	0.16 v 0.13	2.73 v 2.80
mpeg2decode	13 v 18	5.37 v 8.02	0.53 v 0.88	1.24 v 1.05	1.03 v 1.04
pegwit	10 v 11	4.64 v 4.65	0.94 v 0.94	9.25 v 9.25	1.07 v 1.07
gzip	5 v 15	35.85 v 38.87	8.41 v 9.38	0.03 v 0.06	1.50 v 1.51
cpp	29 v 35	7.46 v 7.75	9.28 v 9.81	11.43 v 12.08	1.26 v 1.25

Table 9.4: Result of pickier kernel selection (picky v original). With pickier kernel selection, only kernels with total expected benefit exceeding 5000 cycles and per-entry benefit exceeding 25 cycles were chosen.

for configuration cache. The simplest approach is to eliminate some of the less beneficial kernels (revert them to software execution) using an arbitrary threshold. In one simple experment the threshold for hardware mapping was raised to require an estimated benefit of at least 5000 cycles overall and at least 25 cycles per kernel entry for hardware mapping. The results versus original are presented in Table 9.4. Many kernels are eliminated in with some benchmarks; while much overhead is eliminated, benefit was also removed in some cases. This simple winnowing approach is only moderately effective and in some cases hurts performance. Better approaches would consider call graph and loop structures to predict sets of interfering kernels and reduce them to a size appropriate for the expected configuration cache size. [LCD+00] describes an even more aggressive approach that utilizes a complete trace of kernel entries, allowing complete prediction of cache interference under different winnowings for that execution trace. This last approach's weakness is over-specialization in cases where the actual execution differs greatly from the trace used for compilation.

On the other hand, Garp could be modified in obvious ways to reduce configuration cache misses. The capacity of the configuration cache could of course be increased by some degree;

Benchmark	Speedup vs. normal	Speedup vs. SUIF	% HW Compute	% HW Overhead	# Kernels	# Kernel Executions
go	1.14	0.96	6.13	5.45	96	525491
m88ksim	1.00	1.06	18.15	1.52	15	152221
gcc	1.01	1.00	2.11	1.25	147	112796
compress	1.00	1.04	17.53	7.48	5	654763
li	1.00	1.01	0.01	0.00	3	185
ijpeg	1.00	1.09	6.41	3.26	42	333691
perl	1.00	1.01	0.55	0.18	8	10659
vortex	1.00	0.99	0.16	0.14	20	10644
wavelet-image	1.00	2.80	66.74	14.43	11	8892
mpeg2decode	1.00	1.04	8.02	0.74	18	129908
pegwit	1.01	1.08	4.67	0.43	11	46498
gzip	1.00	1.51	38.87	9.38	15	30110
cpp	1.04	1.31	8.07	6.25	35	41128

Table 9.5: Benchmark execution on Garp with 128-level (effectively infinite) configuration cache.

increasing the size from four to eight planes is estimated to add 15% to the array size, but 128 planes is somewhat unrealistic. Another approach would be to make the configurations smaller. Compressing the configurations might be useful; the distributed configuration cache would likely still store the decompressed representation, but at least the impact on the L2 cache would be reduced. Another approach would consider coarser-grain architectures that need fewer total configuration bits to specify a 32-bit module.

To guage the potential benefits of improving configuration cache performance, the benchmarks were rerun with a simulated Garp array having 128 cache levels. The results are shown in Table 9.5. Performance improved, particularly for ‘`go`’, but there were not significant improvements across the board.

9.3.3 Achieved instruction-level parallelism

Even with a 128-level configuration cache, performance was not impressive. To further investigate, instruction level parallelism of each kernel was studied. Could the lack of performance be due to a lack of instruction level parallelism, which was expected to be a main source of benefit?

Counting operations ILP can be calculated in many alternative ways. ILP is usually expressed as operations per cycle, but that simple definition is imprecise and ignores many subtleties. Given `garpcc`'s strategy of predicated and speculative execution, it could be argued that simply counting all executed operations artificially inflates the ILP compared to nonspeculative software execution in two ways.

The first argument is that many operations are overhead for support this execution style, specifically predicate calculation operations and muxes. However, this is balanced to some degree by the fact that branches internal to the kernel have been eliminated while they count as operations in software execution. In addition, other artifacts of von Neumann execution, e.g. register-to-register moves and unconditional branches, are not counted with Garp array execution while they would be with processor ILP.

Secondly, should an operation executed speculatively be counted during all iterations, or only those when it would have been executed with non-speculative execution?

The right metric depends on the point of view. Approaches to enhancing ILP typically encounter diminishing returns. A greater increase in hardware support for ILP is required to support a smaller increase in effective ILP and performance. Considering specifically ways to count operations in the DFG, one extreme that measures utilization of hardware resources without concern for operation inflation is simply:

$$OPS_{HW} = \sum_{N \in ACTIVE} HWweight_N$$

with

$$HWweight_N = \begin{cases} 2 & \text{if N is a queue access} \\ 1 & \text{otherwise} \end{cases}$$

Queues are weighted as two operations to account for both the memory access and the address increment.

At the other extreme, operations in the DFG can be counted in a way to imitate software non-speculative execution. Operations are weighted so that they effectively they are only counted for those iterations in which they would have actually executed. Extra operations introduced to

support predicated execution are ignored by setting their $weight$ to 0. Comparisons are weighted to 2, however, to include the anticipated cost of associated conditional branches that would be present with software execution. To avoid double counting, Exit nodes are weighted to 0. The equation given these considerations is:

$$OPS_{SW} = \sum_{N \in ACTIVE} execfrac_N \times SWweight_N$$

with

$$SWweight_N = \begin{cases} 0 & \text{if N is predicate logic or multiplexor} \\ 0 & \text{if N is an exit} \\ 2 & \text{if N is a queue access} \\ 2 & \text{if N is a comparison} \\ 1 & \text{otherwise} \end{cases}$$

For simplicity, OPS_{HW} is used to count the operations in a kernel for the following studies. This provides a measurement of the utilization of the coprocessor's resources even at diminishing returns.

ILP range Many kernels achieve a large part of their ILP through pipelining. The ILP achieved once the pipeline reaches steady state is calculated by the equation below. Recall that II is the initiation interval, the delay between the starts of successive iterations with pipelined iterations.

$$ILP_\infty = \frac{OPS}{II}$$

Since ILP is lower during the prologue, overall ILP for the loop will asymptotically approach this value when the number of iterations is large, hence the name ILP_∞. At the other extreme, if the loop executes just one iteration, pipelining does not contribute to ILP, so the ILP is simply the ILP within the hyperblock—the number of operations divided by the schedule length:

$$ILP_1 = \frac{OPS}{SL}$$

The actual ILP during array usage will be somewhere between these bounds, depending on how many iterations a kernel executes per entry, either considering a specific kernel entry or averaged across all entries. The equation for the net ILP is calculated by dividing the total executed operations by the total time for both prologue and steady state execution (ignoring cases where the final iteration exits early), where $ITERS$ is the average number of iterations per kernel entry. This form of the equation charges the filling of the pipeline to the first iteration, then the incremental time to each subsequent iteration:

$$ILP_{net} = \frac{OPS \times ITERS}{SL \times 1 + II \times (ITERS - 1)}$$

The following rearrangement breaks out the pipeline refill penalty $SL - II$ explicitly:

$$ILP_{net} = \frac{OPS \times ITERS}{II \times ITERS + (SL - II)}$$

Figure 9.1 and **Figure 9.2** provide information about the potential ILP range of each kernel as well as the actual ILP_{net} achieved for an average execution. The range bar spans from ILP_1 to ILP_∞, indicating how much benefit can be derived from pipelining. The net ILP indicated by the diamond depends on the average number of iterations per kernel execution but is also influenced by the ratio between the kernel's II and SL; the greater the ratio, the more iterations are required to approach ILP_∞. Again, the ILP indicated in the figures is 'high quality' in that it does not count register-to-register moves or unconditional branches.

The following observations can be made:

- In most cases ILP_1 is modest, often less than 2 and rarely exceeding 5 even with hyperblock formation. However by the way of counting here, the single-issue MIPS core would exhibit an ILP of less than 1 due to unfilled load delay and branch delay slots, register-to-register moves, and unconditional branches.

- In many cases ILP_∞ is much greater than ILP_1, indicating great benefit from pipelining.

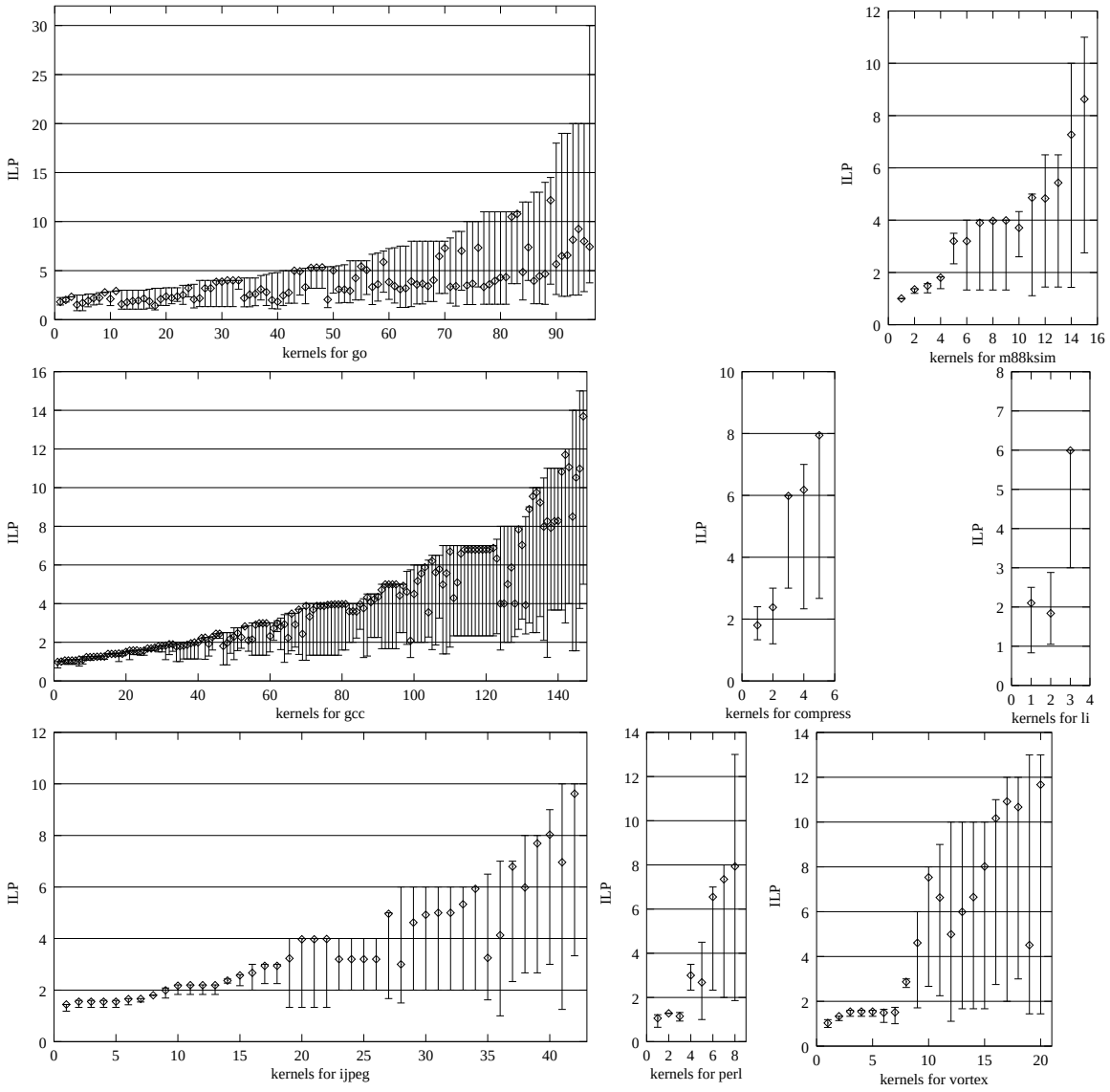


Figure 9.1: ILP_{net} plotted for each kernel on the range of possible instruction level parallelism from ILP_1 (single iteration ILP, i.e. no pipelining) to ILP_∞ (asymptotic pipelined performance). These numbers ignore stalls and any overhead for using the array.

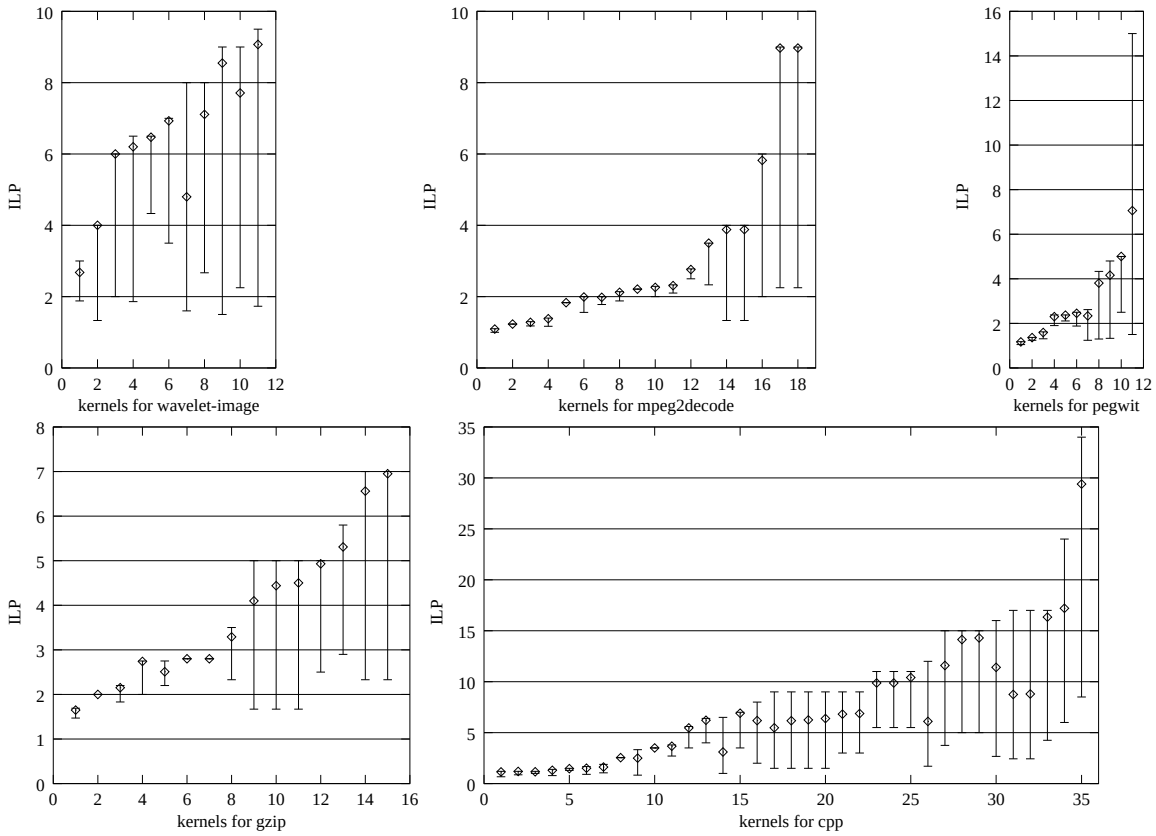


Figure 9.2: ILP_{net} plotted for each kernel on the range of possible instruction level parallelism from ILP_1 (single iteration ILP, i.e. no pipelining) to ILP_∞ (asymptotic pipelined performance). These numbers ignore stalls and any overhead for using the array.

- Kernels vary over the entire range whether their achieved ILP_{net} approachs ILP_{∞} or is instead near ILP_1. While results are typically mixed even within an application, their are some trends. Multimedia benchmarks mpeg2decode and wavelet image compression both approach ILP_{∞} for most kernels. More surprisingly, `gcc`'s results, while mixed, also indicate that many kernels achieve a significant fraction of ILP_{∞}. The worst case in terms of achieving potential, however, was '`go`', where most kernels achieved an ILP_{net} of less than 5 even though the majority of kernels had ILP_{∞} of greater than 5. This indicates that most loops in '`go`' were short-running, either because the original loops were short-running or because kernel pruning significantly increased the number of exits.

- In most cases the achieved ILP_{net} is 8 or less, so would an 8-issue VLIW be just as effective? There are many reasons why that conclusion would be premature. (i) In order to achieve the ILP_{net} shown, the hardware must have resources to support ILP_{∞}. (ii) The ILP is averaged across different cycles when $II > 1$. With the Garp array one or more cycles might execute many more operations than the average ILP. The per-cycle constraint for VLIW may well impact the critical cycle, reducing net ILP. (iii) Unlike Garp, VLIW processors typically have other resource limitations on which combinations of operations can execute in a given cycle, particularly branches. (iv) VLIW execution would likely require the addition of register-to-register moves and loop-closing branches not considered in the counts, increasing required ILP support.

- In summary, the problem is not a severe lack of ILP. Instead, the lack of significant performance improvement must be attributed to (i) short running loops in some cases; (ii) per-entry overhead for using the array (not factored in to the ILP calculation), and (iii) the loops that achieve large speedups do not make up a significant fraction of the application's execution time for many of the studied benchmarks.

Some individual kernels experienced slowdown compared to software even with the large configuration cache. This is somewhat troubling since that simulator configuration matched the model that the kernel pruning and selection algorithms were targeting. Was there more overhead

than expected? This is examined next.

9.3.4 Per-entry overhead

Obviously per-entry overhead can greatly impact the performance of short-running kernels. Study of overhead is useful for two reasons: (i) trying to find ways to reduce it to increase the number of beneficial kernels and to get better performance out of existing kernels, and (ii) finding inaccuracies in overhead estimation, since underestimating the per-entry overhead would cause the bad decision leading to the use of the array when its net performance with overhead is worse than that of software.

Examination of under-performing kernels revealed the following causes.

- **Cache misses** One particularly under-performing kernel experienced a very high percentage of I-cache and L2 cache misses for the instructions setting up use of the array. The I-cache misses cannot be attributed to conflicts with array configurations since configurations never enter the I-cache. Also, the code size of the interface instructions roughly matched the size of the original software loop. The cause in this particular case is likely a change in code layout that leads to a I-cache conflict with some other code that executes between consecutive uses of this kernel. This affect could be expected to even out on average—for every conflict created, one might be removed. However, this case was exacerbated by the additional L2 cache misses. This could be blamed on configurations competing with data and instructions in the L2 cache. Static code size (including configurations) increased by an average of 11% over all the benchmarks. Although not investigated, this indicates that reducing the impact of configurations on the L2 cache by compressing configurations and/or reducing the number of kernels by eliminating those with marginal benefit could help performance by reducing the L2 miss rate.

- **Exit inefficiency** `Garpcc`'s means of implementing multiple kernel exits can be inefficient. Multiple or even all exits can be scheduled for the same cycle in array execution. Each exit module retains an echo of the Boolean value triggering the exit. After exit of a kernel with

multiple exits, these values are checked by the MIPS processor to determine which was the actual one taken. Consider three exits ordered as X, Y, and Z and all scheduled the same cycle in the array. The MIPS processor first checks X, and if it not activated, then checks Y. If Y is not activated then Z can be assumed to be the taken exit. The inefficiency is due to the fact that the order of checks cannot be rearranged to favor more frequently-occurring exits. Even if exit Z is much more frequent than exit X, the MIPS processor must always check X and Y as well, since even if Z is active, it is not the taken exit unless both X and Y are inactive. One means of correcting this would be to use fully-resolved predicates in the array so that the conditions for X and Y are factored into calculating the condition for Z, so that Z is activated only when it truly is the taken exit. This would allow reordering of checks, although it adds some Boolean computation and routing to the datapath; perhaps fully-resolved predicates could be applied selectively only in cases likely to benefit. An even more ambitious approach would make the datapath responsible for computing an integer number in a designated row indicating which exit was taken. Then the MIPS processor could retrieve just the single value to use as an index into a jump table to branch to the appropriate continuation point.

- **Input replication** The overhead estimation assumed that one MTGA (move to Garp array) instruction was needed for each input (kernel-invariant) value. But the synthesis tool GAMA might replicate the value source requiring moves of the same value to registers in multiple modules. A conservative estimate would instead assume that a MTGA instruction is needed for each consumer of the value (determined by fanout from the Input node in the DFG). But that would not be accurate either. The compiler could be smarter and utilize a different version of the MTGA instruction that allows multiple rows to latch the value during a single transfer. But that would complicate synthesis since each receiving row uses the control block for that purpose, preventing it from being involved in memory operations.

A special contribution to per-entry overhead is queue utilization, which is considered in the next subsection.

9.3.5 Reconsidering queue usage

Many short-running kernels had reasonable ILP but too much overhead to provide a net benefit. In such cases the initialization overhead for queues might outweigh the benefits; not using queues might lead to a beneficial kernel.

The benefits of queues and pipelining overlap to a great degree. The queues can in fact be considered to be special hardware support for pipelining memory accesses. Also, the latency benefit from utilizing queues for loads has little direct impact on pipelined performance since a load that could be converted to a queue load cannot be in a cycle and thus cannot affect the recurrence-limited minimum II.

But while the benefits overlap, the overhead for the two approaches is additive. Thus kernels with a low number of iterations, where performance is more sensitive to overhead, might do better without queue use.

Thus it might be reasonable to avoid queue use when the average iteration count is low. One benefit of queue utilization, the elimination of a random access and reduction of resource-limited II by 1 or 2 cycles, has little impact anyway with low iteration counts. But another benefit, reduced area and therefore configuration time, might be more important for short-running queues if configuration time is a significant contributor to overhead for using that kernel.

Examination of kernels with and without queues utilization enabled revealed an important factor: queue utilization can reduce the additional area for pipelining. Consider a kernel where the loop index has two uses: it indexes an array load, and it feeds the comparison that determines loop exit. With no queues, the index calculation must occur early in the schedule to feed the load. A chain of registers or Holds must be added to convey this value to the comparison/Exit which is towards the end of the schedule. But when queues are employed, the index is not needed for the address computation, and so the index increment loop can be moved towards the end of the schedule, eliminating the long register/Hold chain and its area. The net effect is that not using queues can result in some beneficial kernels not fitting, or other kernels being larger than necessary and requiring more configuration overhead.

As an experiment, `garpcc` was given a command-line option to use queues only in ker-

nels where the expected iteration count was above the given threshold. With the threshold set at 5 iterations, the results were mixed and inconclusive, which is not surprising given the multiple conflicting effects. This indicates more factors needed to be considered by the heuristic.

9.3.6 Execution time breakdown

Even though the 32-row Garp array would be very small using a contemporary silicon fabrication process, it would be easier to justify its inclusion if it were useful in accelerating a substantial portion of many benchmarks. Given that the compiler cannot effectively utilize the array for significant portions of some/many benchmarks, the next question is, why? To help answer this question, execution time in each of the following categories is estimated. The breakdown will be presented as fractions of original software execution. The following classification is utilized:

- `os-kernel` time spent in the OS kernel
- `library` time spent in the C library
- `straight` instructions outside of any loop
- `outer` instructions inside of a loop that has an inner loop but which are not in the inner loop itself.
- `swloop` instructions inside an inner loop from which no hardware kernel was formed
- `hw` instructions
- `excluded` instructions are those executed in software after an excluded-path exit.

The numbers are meant for rough guidance only; there are major sources of inaccuracy. Firstly, the breakdown among user code categories (not kernel and not library) is based on estimates made from the SUIF representation, not actual execution time. However, the ratios among kernel, library, and total user code are determined by actual simulations. Secondly, the breakdown does not account for procedure call overhead due to register saves and restores because those instructions are not explicit in the SUIF representation and thus are not estimated.

	hw	straight	outer	excluded	swloop	library	os-kernel
go	16.23	30.35	0.20	1.12	52.03	0.05	0.02
m88ksim	30.71	51.24	0.01	0.02	14.47	3.24	0.30
gcc	6.82	39.11	3.48	0.12	38.59	5.04	6.84
compress	28.27	29.21	8.33	0.00	33.94	0.17	0.08
li	0.02	36.96	1.07	0.00	58.94	0.24	2.77
ijpeg	20.79	3.66	0.56	0.48	72.24	1.47	0.80
perl	2.35	25.63	0.34	0.00	12.21	14.68	44.79
vortex	0.70	84.98	0.06	0.01	4.49	9.47	0.29
wavelet-image	94.91	0.01	0.08	1.07	2.72	0.96	0.25
mpeg2decode	13.60	20.66	0.15	0.00	65.53	0.02	0.03
pegwit	4.71	87.59	0.00	0.00	5.57	1.78	0.35
gzip	67.94	8.06	4.38	0.97	17.61	0.86	0.18
cpp	39.52	5.13	1.41	1.20	39.29	12.95	0.50

Table 9.6: Breakdown of original software execution time by category.

The large fraction of time spent in the OS kernel by '`perl`' was due to floating-point emulation for a floating point register save/restore in the `eval()` function. A large part of the remainder was spent in straightline code, which also occurred in the `eval()` function, which calls itself recursively. The time spent in the library was largely due to `qsort`, which calls the '`perl`'-provided comparison function `sortcmp` which also contributes to straightline code execution time. Other significant library calls include `memmove` and `memcmp`.

Array utilization could be increased by recompiling the C libraries with `garpcc` or even hand-coding them. Array use by the OS kernel is trickier since there may be an active user configuration on the array at the time the OS kernel is entered, requiring an explicit save and restore of the configuration and state. This would probably not be worthwhile except during context switches which require a save of any active user configuration anyway.

Straightline code occurring in leaf procedures could potentially be moved to the 'hw' category through procedure inlining, considered further in Subsection 9.3.8. Transforming recursion to iteration could also move execution time from straightline code to potential 'hw' usage. Outer loops might be attacked by splitting them via loop fission (loop distribution) when possible to separate the part(s) with the inner loop(s); the other parts of the split outer loop then become inner loops.

The ‘swloop’ category includes all inner loops from which no successful kernel was formed. There were many reasons, each of which would be attacked in a different manner. This is considered next.

9.3.7 Breakdown of loops not accelerated

The category ‘swloop’ in the previous subsection is very general in that there can be many reasons that an inner loop cannot form a kernel with a net benefit. In fact even for a single loop there may be many contributing factors preventing beneficial use of the array, complicating the accounting. A natural loop with moderately low average iterations per entry may then have some paths pruned due to infeasible operations and then have additional paths pruned for fitting to available resources. If the sum of the exits is too large (corresponding to low iteration count per kernel entry), the kernel is unsuccessful, but what is to blame? Even when the cause is limited to infeasible operations, there can be different infeasible operations on different paths or on the same path; again, how should the blame be allocated?

The following simplified categories are used. A loop is classified by the first category that matches its characteristics.

- `all-infeasible` The loop had infeasible operations on all paths. Even if the loop is very big, it is considered to belong to this category.
- `low-iters` The original loop (before pruning) did not execute enough iteration per entry on average to overcome the per-entry overhead.
- `low-total` The original loop (before pruning) was not used enough times in total to overcome the assumed one-time configuration overhead.
- `many-infeasible` The loop had infeasible operations on a fraction of paths high enough to increase the number of exits to make the kernel non-beneficial.
- `size` Pruning for size caused the average number of iterations per entry to drop too low to overcome per-entry overhead. Or in another case, fit-pruning failed because it reached a point

	total	low-iter	low-total	size	vmult	div-rem	float	call	misc
go	52.03	13.65	0.67	1.11	1.38	0.03	0.00	32.86	2.34
m88ksim	14.47	5.94	0.00	0.00	0.20	0.00	0.00	0.76	7.57
gcc	38.59	4.39	0.67	0.67	0.79	1.34	0.00	23.27	7.46
compress	33.94	0.00	0.00	0.39	0.00	0.00	0.00	33.55	0.00
li	58.94	12.10	0.00	6.20	0.00	0.00	0.00	14.20	26.44
ijpeg	72.24	1.02	0.01	36.81	24.08	0.00	0.00	9.85	0.46
perl	12.21	0.09	0.02	0.00	0.35	0.00	0.00	8.25	3.50
vortex	4.49	0.25	0.01	0.00	0.00	0.89	0.00	2.90	0.44
wavelet-image	2.72	2.71	0.01	0.00	0.00	0.00	0.00	0.00	0.00
mpeg2decode	65.53	0.75	0.00	12.27	23.70	0.01	0.00	1.39	27.40
pegwit	5.57	0.22	0.09	3.38	0.00	0.00	0.00	1.88	0.00
gzip	17.61	0.41	0.04	0.92	0.17	0.00	0.00	16.06	0.00
cpp	39.29	1.75	0.02	31.81	0.04	0.00	0.00	5.03	0.63

Table 9.7: Percentage of execution time spent in loops that could not be accelerated using the array.

where no remaining prune edges remained, yet the kernel was still too big.

With categories `all-infeasible` and `many-infeasible`, further information as to the nature of the infeasible operation(s) is described as below:

1. Only infeasible operations from basic blocks executing in over 10% of the total iterations are considered.

2. All infeasible operations types from such blocks are considered.

3. If all such operations the same type, then that is the infeasible cause for the loop. If there are different types of infeasible operations, then all of the loop's execution time belongs to the `misc` category. For example, the `misc` time in the '`mpeg2dec`' benchmarks is due to a loop containing both `call` and `vmult` (variable $\times$ variable) infeasible operations.

Besides containing loops with mixtures of infeasible operations, the `misc` category also contains loops infeasible due to compiler "builtin" functions. The `misc` time for the '`li`' benchmark is due to `varargs` calls which are treated as a compiler builtin.

Additional observations:

- Almost all of the divisions and remainders were by powers of two. But since they were operating on signed values, they could not be strength reduced to simple shifts or masks. They could however be reduced to a slightly more complicated computation than just a shift, involving a correction using the sign bit, but this was not implemented. A much more ambitious approach to strength reduction of division and remainder in loops is described in [SLGA91], but this was not pursued either. In many of the cases the values were obviously always non-negative and thus the programmer could correct this by simply using the `unsigned` type rather than `int`, which would allow simple strength reduction. Value range analysis such as [BGWS00] might be useful for automatically converting such cases.

- With the exception of '`mpeg2decode`' and '`ijpeg`', variable multiplications are not a significant source of infeasible operations. The approach described in Section 6.6 with associated transformations would remove many of the infeasible multiplications in '`mpeg2decode`', although the loop likely could still not utilize the array without further transformation due to its large size; loop fission, not implemented, would be applicable and could lead to beneficial kernels. Alteratively, restructuring of the source code could result in mostly feasible kernels with no compiler modification. The infeasible multiplications in '`ijpeg`' would be more difficult to remove. The ones contributing most heavily to 'swloop' time are products of a true variable with a kernel-invariant value; only run-time specialization could convert these to feasible constant multiplications.

- Procedure calls prevent loop acceleration from a moderate to great degree in most benchmarks. Procedure inlining has the potential of reducing the execution percentage not only in the `swloop-call` category but also in the `straight` category in Table 9.6. Experience with inlining is described in Subsection 9.3.8.

9.3.8 Effect of procedure inlining

The large percentage of non-accelerable "straightline" and "swloop-call" code noted in the execution time breakdown could be due to a large amount time spend in loopless leaf subroutines

called from within high-count loops. If this is the case, procedure inlining could provide great benefit. After inlining, the loop that formerly contained the call may become a potential hardware kernel that includes the execution time formerly spent in the leaf procedure.

In `garpcc`, a simple procedure inlining pass was implemented. After disappointing results, however, it was turned off for the results presented in this thesis.

The simple inlining pass looked for two situations:

(i) when the callee is leaf and loopless *and* the call site occurs within a loop (addressing the situation above).

(ii) when the callee is leaf, contains at least one loop, *and* the call site contains at least one constant parameter. The reasoning is that the constant parameter gets propagated through the loop(s) of the callee, leading to additional optimization—specializing the kernel(s) to each call site.

If inlining were performed after profiling, inlining could be limited to those call sites having significant execution counts. But then profiling would need to be performed again on the transformed code to get accurate profiling counts at each inline site. So `garpcc` instead performs inlining before profiling, accepting the code expansion costs associated with inlining in cases that are not significant in terms of execution counts. While code expansion would be a very important consideration in an embedded context, it is not so important in Garp's assumed desktop role.

The individual contributing benefits and costs involved with each of the two inlining cases can be predicted, although the net effect requires experiment. In both cases, code expansion is a drawback. The analysis of case (i) is straightforward: without inlining, neither the callee nor the loop containing the call site would have been accelerable, so no harm is done. Case (ii) is more complicated. Before inlining there would be a single kernel; afterwards the potential kernel occurs in multiple places. The main cost of inlining is that each new kernel is distinct, and thus multiple rather than one configuration must be loaded at least initially, also possibly putting more pressure on the configuration cache. The benefits arise from the opportunity to specialize the kernel for each call site. Constant parameters are propagated through the inlined code, potentially creating many optimization opportunities, although it is not guaranteed that any kernel will benefit. Also, when

the same procedure is inlined in multiple locations, the kernel at each inline site can be pruned to optimize for the site's profile information. Without inlining, the single original kernel's pruning could optimize only for the weighted average. Similarly, the decision of whether to revert to software can be made individually for each inlined instantiation of the kernel. A more sophisticated (but unimplemented) approach would trace the flow of the parameters within the callee so that the inlining is only performed when the constant parameter(s) are guaranteed to impact at least one loop in the callee.

Initial experiments with inlining did increase the percent of hardware execution time for most benchmarks, although only very modestly in most cases. In the two benchmarks starting with the highest percentage of straightline code unfortunately benefited the least from inlining. Closer inspection of the two benchmarks revealed some limitations of inlining as implemented by `garpcc`:

- **Separation between loop and leaf** The simple inlining rules fail to recognize some fairly straightforward cases. For example, when a loop in procedure X contains a call to straight line procedure Y which in turn contains a call to straight line leaf procedure Z, no inlining is performed. Y is not inlined into X because Y is not leaf. Y would be leaf if Z were inlined into it, but Z is not inlined to Y because as seen from the viewpoint of Y, the call to Z does not occur within a loop. To correct this weakness, the inlining algorithm would need to propagate information around the call graph, as opposed to the current approach which only considers information available locally in each caller-callee pair.

- **Calls on rare paths** Vortex started with 85% straightline code, and yet inlining was not performed at even a single call site. That is because Vortex is written so that most subroutines contain a conditional call to an error reporting subroutine, which in turn contains a call to the sprintf() library routine. No routine containing this construct can be inlined, even though the error reporting subroutine is rarely if ever called. This is exactly the Garp hyperblock hypothesis—that difficult cases typically occur on rarely-taken paths. If such a straightline subroutine were inlined into the loop, it would be the classic Garp hyperblock case: the difficult case (subroutine call) would occur on a rarely-executed path that could be excluded.

If profiling information were available, a procedure could be classified as “mostly” leaf if its only procedure calls occurred on rare paths.

- **Recursion** The Perl benchmark spends most of its execution time in the recursive procedure `eval()`. In this case simple inlining cannot creating a loop that can be accelerated.

- **Not considering size of callee** If a kernel affected by inlining is too large for hardware implementation on available resources, again, the desired benefit is not realized. However, a large loop may eventually—after optimization and pruning (hyperblock formation)—yield a small implementable hardware kernel that gains great speedup. But pruning cannot even be estimated during inlining since there is no profiling information, so simply considering the size of the callee does not give much information about whether or not inlining will enable further hardware execution. However, in any case size estimates could be used to weigh against code expansion, since even pruned paths are realized as replicated software code at the inline site (as tails after hyperblock exits). If profiling were available, the adjusted size of the callee could be estimated from the computation on just the common paths.

After these observations, it seems clear that inlining should instead be performed after profiling. Performing inlining that late in the compiler flow has its drawbacks: (i) nonspecialized profile data at each call site; (ii) nonspecialized memory dependency analysis; (iii) more bookkeeping during inlining to maintain or update previous analysis annotations. Yet, changing the compiler flow to place inlining closer to hyperblock formation—maybe even integrating the two—seems to promise many more benefits than penalties. Ideally, profiling would guide inlining, and then be performed again afterwards to gather specialized profiling counts. While performing multiple profiling runs would be tedious using manually directed profiling, it would fit naturally into a continuous feedback-directed optimization context.

Chapter 10

Conclusion

10.1 Summary

The Garp architecture was designed with a reconfigurable coprocessor for accelerating loops from general-purpose programs. The spatial form of computation allowed by the coprocessor has many benefits for computation with the right characteristics; in particular large amounts of parallelism can be exploited.

The challenge for automatic compilation is finding parallelism in the sequential C source code. Simple if-conversion can convert any entire loop from microprocessor style control flow to fully speculative, predicated computation expressed in a dataflow graph ready for implementation using a spatial computing platform. But this approach does not often work with loops from real general-purpose applications. Loops often contain operations that are 'infeasible' for implementation using the coprocessor and/or are too large for complete implementation using the array.

`Garpcc` borrows the hyperblock structure from VLIW compilation to address these problems. The hyperblock includes just the often-executed paths through a loop. This allows exclusion of rare paths that contain infeasible operations that would otherwise make the entire loop infeasible for hardware acceleration. Paths can also be excluded for the purposes of fitting the kernel to available resources and/or to increase performance.

The DFG representation of the remaining computation allows straightforward, efficient

implementation of many common optimizations; in the DFG, control flow has been eliminated, all data definition-use relationships are explicit, and computation excluded from the hyperblock does not interfere with the optimizations. One particular instance is recognition of memory accesses that can be converted to use Garp's queues; the required analysis of address strides, predicates, and dependence are all straightforward with the DFG representation.

From the DFG a fully spatial optimized datapath is constructed. The spatial approach allows mapping of multiple operators into optimized modules on the reconfigurable coprocessor. The mapping algorithm simultaneously considers relative placement of modules and works to optimize the critical path/cycle. The spatial datapath is also amenable to pipelined execution, greatly increasing peak throughput. Module scheduling is used to create a pipelined schedule that avoids inter-iteration conflicts for the memory port.

In experiments compiling large benchmarks, `Garpcc` was able to expose large amounts of peak ILP in many loops. However, with short-running loops, net ILP was significantly less than peak ILP because (i) overhead for using the array was significant, and (ii) with short-running loops little or no time is spent with the pipeline full—the state required for maximum ILP. Kernel pruning exacerbated the problem because the extra exits had the effect of reducing average iteration count per array use. However, this was unavoidable; without pruning for size and feasibility, those kernels could not use the array at all. Further effort could be devoted to reduce the overhead introduced by `garpcc`. In one particular situation, because of the overhead in using Garp queues, perhaps they should not be used when the average iteration count is low.

The assumption that the Garp array configuration cache would hold the working set of configurations turned out to be false; high miss rates were observed for some benchmarks. To correct this, the compiler should to perform a global kernel selection phase reverting marginally beneficial kernels back to software so that they do not interfere with more beneficial kernels. Alternately, architecture/implementation changes could be considered to increase the capacity of the configuration cache.

The fraction of time spent using the array was less than would be hoped. The causes of low performance and low array utilization are intertwined: if every kernel achieved better hardware

performance, kernels just below the break-even point for benefiting from the array would become beneficial, increasing the fraction of the application using the array.

Factors specifically contributing to the low fraction of the application that can use the array are listed below:

- The attempted approach to inlining had many shortcomings and so was not used.
- Recursion cannot be handled by `garpcc`. No attempt to convert recursion to iteration was attempted.
- The standard C libraries linked with the executable do not use the array. They should be recompiled using `garpcc` or even hand-optimized.

Intelligent profile-directed inlining appears to be the single most useful approach to increasing the utilization and benefit of the array. Because specialized profile counts after inlining is also desired, a continuous profiling and optimization framework would be very useful.

An important issue not addressed in this thesis is `garpcc`'s dependence on accurate profiling information. Using the array is big gain, big loss proposition compared to most other compiler transformations. Likely any performance gains would not hold up when the actual dataset gives an execution pattern drastically different than that recorded by profiling. Using dynamic runtime hardware/software selection could make the performance more robust in the face of varying datasets but would add extra overhead.

In summary, there is still much room for improvement in `garpcc`, particularly to reduce array usage overhead, to reduce configuration cache thrashing, and to increase the amount of candidate loops by performing intelligent procedure inlining. Perhaps architecture improvements could reduce overhead to allow the Garp array to be more useful for both short- and long-running loops. However perhaps the Garp array is best left for targeting long-running loops while using other means to exploit ILP in short-running loops; for example, the main processor could be replaced with a VLIW or superscalar processor, or the MIPS processor could be augmented with a reconfigurable function unit that has less entry overhead but also cannot achieve the peak ILP of the

Garp array. In any case, the Garp array's footprint is very small when implemented in a contemporary silicon process and so would be worth adding even if not used for a large fraction of execution time. Finally, `garpcc` has assumed the most challenging initial conditions—no Garp-specific help from the programmer. With even minor code restructuring and/or inserted hints, performance would be greatly improved in most cases.

10.2 Related Work

A distinguising charachteristic of the work here is that the starting point is ISO C with no hints or annotations from the programmer. Up to this point the only other work reporting results from automatically compiling large C benchmarks to a reconfigurable architecture was the PRISC project [RS94]. However, the PRISC architecture differed greatly from Garp and so there was little in common with their respective compilation paths. The PRISC architecture featured a programmable function unit (PFU) rather than a loop-accelerating coprocessor as in Garp. PRISC's PFU was integrated into a normal RISC pipeline, executing for just the one cycle of the execute stage. This integration allowed just the two inputs and one output based on the regfile ports. Furthermore the one-cycle timing constraint restricted its structure to 3 32-bit rows of CLBs with no carry chains. These considerations limited the possible uses of the PFU and thus the required compiler analyses and transformations. The phase that identified potential PFU uses considered the RISC object code of the application, looking for short sequences that match a small number of targeted PFU utilizations, for example groups of bitwise-logical instructions or computations based on memory look-up tables. Although a 22% performance benefit was realized across the benchmarks, most of the improvement came from a single PFU optimization in a single benchmark, '`eqntott`', which translates Boolean equations to truth tables.

The PRISM-II compiler [WAL+93, AWG94] was an early effort that also used a DFG as the stepping stone between C and hardware implementation. It targeted a separate microprocessor-FPGA system. The source language was C although the programmer was required to identify the critical sections to be mapped to the FPGA. The framework could map both straightline functions

and functions with loops to the FPGA. A major limitation of the PRISM-II system is that the FPGA had no direct access to memory. No pipelining was performed, but the controller was smart enough to determine which computation paths were selected by the muxes—if all short paths were used, the results for the current iteration could be latched earlier allowing a quicker start to the next iteration.

Maya Gokhale's work compiling from C-like languages to reconfigurable platforms has spanned several projects with many collaborators. To ease programming of the multi-FPGA Splash-2 array [ABD92] she helped develop the dbC language [GM93] which contained support for variable bit width, bit-serial, data-parallel computations. Much work done later with the NAPA-C compiler [GS98, GSG00] targeted the National Semiconductor's NAPA architecture [RLG+98], which combined a microprocessor core, a reconfigurable coprocessor ("adaptive logic processor" or "ALP"), plus special scratchpad memories and an I/O port directly to the ALP. The most recent project is the Streams-C system [GSAK00]. All of these projects include language extensions that must be utilized by the programmer to effectively use the reconfigurable resources.

Of Gokhale's work, the NAPA-C work most closely resembles the `garpcc` work due to the similarities of their target platforms. With NAPA-C user annotations indicate data and code partitioning between the main processor and the reconfigurable coprocessor. Additional research investigated automatic allocation of arrays to NAPA's multiple scratchpad memory banks [GS99]; this has in common with `garpcc`'s queue analysis the goal of allowing parallel accesses to different memory regions when coherence is not required. Unlike `garpcc`, the NAPA-C compiler can generate both looping and non-looping custom macro instructions on the reconfigurable coprocessor; in the non-looping case the synthesis tool MARGE [GG97] can use a more traditional high-level synthesis approach that is not fully spatial and can re-use function units.

Weinhardt's work has investigated pipelined datapath generation using vector compilation techniques [WLar]. It accepted ISO C although user annotations helped in many cases. While much of his intended compiler flow was described, it is uncertain how much was implemented, and only results for small loop nests were reported. It targeted loops containing only regular memory dependences (those resulting from constant-stride array accesses). An important part of his compilation flow is the reduction of memory traffic by reusing data for index-shifted accesses to the

same array, similar to some of `garpcc`'s memory optimizations. Weinhardt's compiler is much more ambitious than `garpcc` regarding loop transformations to increase hardware performance. In addition to loop normalization and complete loop unrolling (both also implemented by `garpcc`), it performs loop merging, loop interchange, partial loop unrolling, and loop tiling (partial unrolling combined with loop interchange). It does not intelligently perform a combination of transforms as with guided unimodular transforms [WL91], but instead either relies on user specified transforms or generates a set of various transformed loops and selects the best.

Both the NAPA-C compiler and Weinhardt's work attempt to pipeline inner loops. They both utilize if-conversion to convert an entire loop body to a single basic block, but neither has the capability to exclude rarely-executed portions of the loop. For those research efforts, exclusion might not be practical because of the assumed high overhead for switching from hardware to software execution and back. Another limitation of both these projects as currently reported is that pipelined configurations can be generated only for counted FOR loops without breaks; in contrast, `garpcc` can pipeline loops with data-dependent exits, including `break`s. Also, those two works handle only constant-stride array-based memory accesses when pipelining. This is mainly due to their target architectures, which do not facilitate low-latency fetches of arbitrary memory locations. Instead they assume all memory access occurs via programmable external DMA engines similar to Garp's memory queues, or to local copies of arrays.

The NAPA-C compiler's approach to pipelining [GSG00] is similar to the Garp compiler in that it is fully spatial and in that it utilizes iterative modulo scheduling to resolve memory scheduling conflicts. However, its target array's clock period is not fixed but variable, determined by the latency of the longest operator, so it is not clear whether memory ports are always utilized at their full capacity.

Weinhardt's pipelined synthesis approach more closely resembles hardware retiming algorithms [LS83], in that it starts by considering the acyclic portion of the DFG to be purely combinational, and then moves in registers to minimize clock cycle time and/or added area. With his approach, the clock period is variable and is set to one per iteration, determined by the longest feedback cycle. Somewhat similar to GAMA's sequencer, Weinhardt's synthesized pipeline includes a

"validity bit" shift register that enables proper state activation during both prologue and epilogue. Since Weinhardt's work considers only counted loops, it can count exactly the correct number of "valid" bits to send down the register; in contrast, `garpcc`'s pipelining scheme with speculative execution does not require anticipation of the end of the loop.

10.3 Future Work

Many improvements could be made to `garpcc`, some of which have already been mentioned.

Optimized libraries

In the experiments, the benchmarks were linked with C libraries that utilized just the MIPS core. A real production system would have C libraries at least automatically compiled with `garpcc` but more likely hand-coded to fully exploit the Garp array. Hand-coded libraries would have the advantage that the writer could insert tests on the arguments to determine when it was better to use the array and when it was better to use the software version of the library routine (see also "Dynamic hardware-software selection" below). However, C library use of the array could complicate a global kernel selection algorithm that tries to minimize thrashing in the configuration cache.

Profiling-based procedure inlining

As described in Subsection 9.3.8, a smart inlining pass, likely combined with kernel formation, could lead to many more and/or better performing kernels.

Memory optimization: guarding stores to allow reordering

Currently, a store and a loop exit must be kept in the original order. This can be relaxed when the exit condition is calculated long before the exit is actually executed; the condition (i.e.

predicate) can be used to guard the store. As expressed in C code, the computation would be transformed from this:

```
if (a<b) break;
*x = a;          /* can't be reordered as is */
```

to this:

```
p = a<b;
...
if (!p) *x = a;  /* moved earlier since it's guarded */
...
if (p) break;
```

Essentially, this optimization would selectively convert the default partially resolved predicated form to fully resolved predicate form. The increased scheduling flexibility comes at the cost of extra routing of predicates.

Loop unrolling in `garpcc`

`Garpcc` does not currently consider loop unrolling. There are some situations where it could lead to a substantial performance improvement in combination with pipelining [LH95].

Other loop transforms

Many loop transformations from vector and parallel compilation would be useful. Examples include loop interchange, reversal, and skewing. One goal would be to increase the number of iterations in the inner loop. Another goal would be to increase the number of memory accesses having unit stride to enable more uses of Garp's queues.

Loop fission and fusion could also be useful. Loop fission—splitting one loop into two—could transform one kernel that is too large into two that do fit into the reconfigurable resources. On the other hand, if loop fusion can be applied to create a new larger kernel that still fits, kernel overhead will be reduced.

Many of these transforms were considered by Weinhardt [WLar]. His applications were typically nested loops containing regular array accesses. It is not certain which if any of the transformations would find application in general-purpose code.

Dynamic hardware-software selection

Currently the compiler has to decide at compile time whether a loop should be executed always on the reconfigurable array or always in software; it uses profiling information and performance estimates to judge which will likely give the best performance. However, sometimes profiling information is not available; sometimes there are wide variations in how many iterations a loop goes each times it is entered. Both cases might benefit from the addition of a small amount of decision-making code that can use run-time information to make a better guess on the HW/SW decision. For some loops the iteration count can be deduced; this has been implemented but did not give noticeable improvement.

To attack loops where the iteration count is not obvious, *predictors* could be employed. The most obvious predictor is to remember how many iterations the loop executed last time and uses this to make the decision for the next time (typically, more iterations means HW is better, fewer means SW is better). This is analogous to branch prediction techniques that use the past behavior to try to predict the future. It is not clear whether it would be accurate enough to give benefit considering the added costs for collecting data and making the decision each loop entry.

Small vector data packing (ala MMX/VIS) in `garpcc`

The Garp array can support segmented datapaths of 8 and 16 bits, so an obvious extension to `garpcc` is support for construction of such datapaths either automatically or under programmer guidance.

Dynamic scheduler

The current Garp compilation approach, where every iteration has the same (worst case) schedule, can be inefficient. The worst case is when there are multiple paths with dependency chains of greatly uneven length, and pipelining is not possible. Every iteration has to wait for the longest path to complete, even those iterations when those computations are thrown away. This project would make the Garp configurations a bit smarter, so that if it knows that the short path was

taken, it can start the next iteration earlier (similar to the PRISM-II controller [AWG94], although that case did not have to worry about scheduling memory accesses). This approach would likely be used for kernels that cannot be heavily pipelined, since schedules that are both overlapping and variable would make it difficult to guarantee that memory accesses do not conflict unless some type of arbitration was performed in the new enhanced controller.

Memory optimization - dynamic address disambiguation

With high-ILP processors like Garp, it is very important to remove as many dependencies between memory accesses as possible. But if no information is known about the addresses of the two accesses—specifically, if it cannot be guaranteed at compile time that they cannot access the same location—then they must be kept in the original program order. However, another approach is to allow the reordering, but check at run time whether the two addresses are the same, and if so, take corrective action. This general approach has been implemented in different specific contexts:

- The IA-64 architecture calls it “data speculation” [Int99] and uses a small associative table of watched addresses to see if any subsequent accesses are to the same addresses and thus make the reordering done by the compiler invalid; when such a violation is detected, a branch is taken to “fixup” code.

- Many other hardware techniques have been proposed, for example “coherent registers” (CRegs) [DO94].

- Software technique for dynamic pointer disambiguation have also been suggested, where explicit code is inserted to do the memory address comparisons [Nic89]. It was found to be beneficial only for very wide-issue VLIW processors in which case the added code did not significantly compete with the original computation for issue slots. This would be the same situation using the Garp array.

With `garpcc` the difference between hardware and software implementation is not so clear-cut; starting from either point the implementation using the array would be similar. Although

the additional computation would not compete for per-cycle issue slots on the Garp array, it would contribute to the overall configuration size, and so would need to be applied intelligently.

Dynamic specialization of configurations

This project would implement a mechanism for actually editing and specializing Garp configurations at run-time (to optimize for data that is known at run time but not at compile time). The obvious optimization example is multiplication; it is expensive to multiply two variables, but cheap to multiply a variable and a constant. Run-time configuration specialization has been studied for some specialized applications such as neural networks [EH94] but not for loops automatically extracted from C programs.

Targeting fixed hardware loop accelerators

Currently the Garp compiler extracts loops for acceleration on the reconfigurable array. An extension would retarget the compiler to System on a Chip (SoC) applications, where accelerated loops are implemented directly as synthesized hardware; thus one program would get compiled to a chip containing the MIPS processor plus some number of synthesized datapaths, one per accelerated loop. The mapping and generation of VLSI modules would be very similar to the approach taken in GAMA. It might be worth considering having the MIPS core, hard datapaths, and also a reconfigurable datapath.

Bibliography

[AAW+96] S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S.-W. Liao, B. M. Murphy, R. S. French, M. S. Lam, and M. W. Hall. Multiprocessors from a Software Perspective. *IEEE Micro*, 16(3):52–61, June 1996. See also http://suif.stanford.edu/.

[ABD92] J. Arnold, D. Buell, and E. Davis. Splash 2. In *Proceedings, 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '92)*, pages 316–322, 1992.

[AG85] A. Aho and M. Ganapathi. Efficient Tree Pattern Matching: An Aid to Code Generation. In *Conf. Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 334–340, January 1985.

[AJLA95] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[AJU77] A.V. Aho, S.C. Johnson, and J.D. Ullman. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM*, 24(1):146–60, January 1977.

[APR+] Matthew Aubury, Ian Page, Geoff Randall, Jonathan Saul, and Robin Watts. Handel-C Language Reference Guide. Available at ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Jon.Saul/handelc.ps.gz.

[Asa98] Krste Asanovic. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, May 1998. Also available as technical report UCB/CSD-98-1014.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Pub. Co., 1986.

[AWG94] L. Agarwal, M. Wazlowski, and S. Ghosh. An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 101–10. IEEE Comput. Soc. Press, 1994. AN4754552.

[BG02] Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[BGWS00] Mihai Budiu, Seth Copen Goldstein, Kip Walker, and Majd Sakr. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *Proceedings Euro-Par 2000*, August 2000.

[BL96] Thomas Ball and James Larus. Efficient Path Profiling. In *Proceedings MICRO-29*. IEEE Press, December 1996.

[CCDW98] Timothy J. Callahan, Philip Chong, André DeHon, and John Wawrzynek. Rapid Module Mapping and Placement for FPGAs. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 123–132, Monterey CA USA, 1998. ACM.

[CFR+91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–90, 1991.

[Cha81] A. E. Charlesworth. An Approach to Scientific Array Processing: The Architecture Design of the AP-120B / FPS-164 Family. *IEEE Computer*, 14(12):12–30, December 1981.

[CL97] K.D. Cooper and J. Lu. Register Promotion in C Programs. *SIGPLAN Notices*, 32(5):308–19, May 1997.

[CSC+00] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Path Analysis and Renaming for Predicated Instruction Scheduling. *International Journal of Parallel Programming*, 28(6):563–588, 2000.

[DeH94] A. DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–9. IEEE Comput. Soc. Press, 1994. AN4754544.

[DGK94] Srinivas Devadas, Abhijit Ghosh, and Kurt William Keutzer. *Logic Synthesis*. McGraw-Hill, 1994.

[DHM+88] T. Diede, C. F. Hagenmaier, G. S. Miranker, J. J. Rubinstein, and W. S. jr. Worley. The Titan Graphics Supercomputer Architecture. *IEEE Computer*, 21(9):13–31, 1988.

[DO94] Peter Dahl and Matthew O'Keefe. Reducing Memory Traffic with CRegs. In *Proceedings of the 27th International Symposium on Microarchitecture*, November 1994.

[EH94] J. G. Eldredge and B. L. Hutchings. Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, Napa, CA, April 1994.

[FH95] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[Fis83] Joseph A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proc. Tenth International Symposium on Computer Architecture*, pages 140–150, June 1983.

[Fra92] Robert Francis. *Technology Mapping for Lookup-Table Based Field-Programmable Ga te Arrays*. PhD thesis, University of Toronto, 1992.

[GG93] S.A. Guccione and M.J. Gonzalez. A Data-Parallel Programming Model for Reconfigurable Architectures. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 79–87. IEEE Comput. Soc. Press, 1993. AN4630527.

[GG97] Maya Gokhale and Edson Gomersall. High Level Compilation for Fine Grained FPGAs. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, April 1997.

[GM93] Maya Gokhale and Ron Minnich. FPGA Computing in a Data Parallel C. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–101, Napa, CA, April 1993.

[GS94] M. Gokhale and B. Schott. Data parallel C on a reconfigurable logic array. *Journal of Supercomputing*, pages 1–24, 1994.

[GS98] M.B. Gokhale and J.M. Stone. NAPA C: compiling for a hybrid RISC/FPGA architecture. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251) Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–35. IEEE Comput. Soc, April 1998.

[GS99] M.B. Gokhale and J.M. Stone. Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00375) Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 63–9. IEEE Comput. Soc, Apr 1999.

[GSAK00] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In B. L. Hutchings, editor, *Proceedings of Eight Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 49–56, Napa, CA, April 2000.

[GSG00] M.B. Gokhale, J.M. Stone, and E. Gomersall. Co-Synthesis to a Hybrid RISC/FPGA

Architecture. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 24(2):165–80, Mar 2000.

[Har77] W.H. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, 3(3):243–50, May 1977.

[Hau00] John Reid Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware*. PhD thesis, University of California at Berkeley, December 2000.

[HS95] Samuel P. Harbison and Guy L. Steele. *C, A Reference Manual, 4th ed.* Prentice-Hall, 1995.

[HT72] R.G. Hintz and D.P. Tate. Control Data STAR-100 Processor Design. In *Digest of Papers of the Six Annual IEEE Computer Society International Conference*, pages 1–4. IEEE, 1972.

[HY97] Glenn Holloway and Cliff Young. The Flow Analysis and Transformation Libraries of Machine SUIF. In *Proceedings of the Second SUIF Compiler Workshop*, August 1997. Available from `http://www-suif.stanford.edu/suifconf/suifconf2/`.

[Int99] Intel. IA-64 Application Developer's Architecture Guide, May 1999. Available at http://www.intel.com/design/ia64/downloads/adag.htm.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[Keu87] Kurt Keutzer. DAGON: Technology Binding and Local Optimization by DAG Matching. In *Proc. 24th ACM/IEEE Design Automation Conference*, pages 341–347. ACM, 1987.

[Koc96] Andreas Koch. Module Compaction in FPGA-based Regular Datapaths. In *Proc. 33rd ACM/IEEE Design Automation Conference*. ACM, 1996.

[KPP+98] S. Kumar, L. Pires, D. Pandalai, M. Vokta, J. Golusky, S. Wadi, and H. Spaanenburg. Benchmarking Technology for Configurable Computing System. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 273–4. IEEE Comput. Soc, Apr 1998.

[KRS94] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Partial Dead Code Elimination. In *SIGPLAN Conference on Programming Language Design and Implemen tation*, pages 147–158, 1994.

[LCD+00] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph E. Harr, Uday Kurkure, and Jon Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proc. 37th ACM/IEEE Design Automation Conference DAC 2000*. IEEE Computer Society Press, June 2000.

[LH95] D.M. Lavery and W.-W. Hwu. Unrolling-based optimizations for modulo scheduling. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (Cat. No.95TB100012) Proceedings of MICRO'95: 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 327–37. IEEE Comput. Soc. Press, 1995.

[LH96] D.M. Lavery and W.W. Hwu. Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture.*, pages 126–37. IEEE Comput. Soc. Press, Dec 1996.

[LS83] C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, pages 41–67, 1983.

[Mah92] Scott Alan Mahlke. Design and Implementation of a Portable Global Code Optimizer. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[Mah96] Scott A. Mahlke. *Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois, Urbana, IL, 1996.

[McK95] S. A. McKee. *Maximizing Memory Bandwidth for Streamed Computations*. PhD thesis, University of Virginia, May 1995.

[MKW+98] S.A. McKee, R.H. Klenke, K.L. Wright, W.A. Wulf, M.H. Salinas, J.H. Aylor, and A.P. Batson. Smarter Memory: Improving Bandwidth for Streamed References. *IEEE Computer*, 31(7):54–63, Jul 1998.

[MLC+92] 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 *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[Nic89] A. Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers*, 38(5):663–78, May 1989.

[Pap94] Christos Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[PL91] I. Page and W. Luk. Compiling occam into FPGAs. In *FPGAs. International Workshop on Field Programmable Logic and Applications*, pages 271–283, Oxford, UK, September 1991.

[Rau94] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture.*, pages 63–74. ACM, 1994.

[Rau96] B.R. Rau. Iterative Modulo Scheduling. *International Journal of Parallel Programming*, 24(1):3–64, 1996.

[RG81] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Proceedings of the Fourteenth Annual Workshop of Microprogramming*, pages 183–198, October 1981.

[RLG+98] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *Proceedings IEEE*

Symposium on FPGAs for Custom Computing Machines, pages 28–37. IEEE Comput. Soc, Apr 1998.

[RS94] R. Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, November 1994.

[RST92] B.R. Rau, M.S. Schlansker, and P.P. Tirumalai. Code Generation Schema for Modulo Scheduled Loops. In *Proc. 25th Annual International Symposium on Microarchitecture*, pages 158–69, December 1992.

[Rus78] R.M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, 1978.

[SLGA91] Jeffrey W. Sheldon, Walter Lee, Benjamin Greenwald, and Saman Amarasinghe. Strength Reduction of Integer Divison and Modulo Operations. In *Proceedings of the '01 Conference on Languages and Compilers for Parallel Computing*, August 1991.

[SMJ99] M. Schlansker, S. Mahlke, and R. Johnson. Control CPR: a Branch Height Reduction Optimization for EPIC Architectures. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, volume 34, pages 155–68, May 1999.

[SS00] Nathan T. Slingerland and Alan Jay Smith. Design and Characterization of the Berkeley Multimedia Workload. Technical Report UCB/CSD-00-1122, University of California at Berkeley, Computer Science Division, December 2000.

[Tho64] J. E. Thornton. Parallel Operation in the Control Data 6600. In *Proc. Fall Joint Computer Conference*, pages 33–40, 1964.

[Tji86] Steven Tjiang. Twig Reference Manual. Comp. Sci. Tech. Rep. 120, AT&T Bell Laboratories, January 1986.

[Tom67] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[WAL+93] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16. IEEE Comput. Soc. Press, 1993. AN4630519.

[Wil97] Robert P. Wilson. *Efficient Context-Sensitive Pointer Analysis For C Programs* . PhD thesis, Stanford University, December 1997.

[Wir98] Niklaus Wirth. Hardware Compilation: Translating Programs into Circuits. *IEEE Computer*, 31(6):25–31, June 1998.

[WL91] M.E. Wolf and M.S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, vol.2,(no.4):452–71, Oct. 1991.

[WLar] M. Weinhardt and W. Luk. Pipeline Vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, to appear.

[Wol89] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman Pub., 1989.

[You98] R. Clifford Young. *Path-based Compilation*. PhD thesis, Harvard University, September 1998.

Appendix A

Real Examples

This appendix contains intermediate files and automatically generated graphics from the `garpcc` compilation of the programs shown.

A.1 Non-pipelined example

This very simple example will be used to illustrate non-pipelined execution:

```
main()
{
int i, j;

        for (i=0; i<10; i++) j += i;
        printf("j = %d\n", j);

}
```

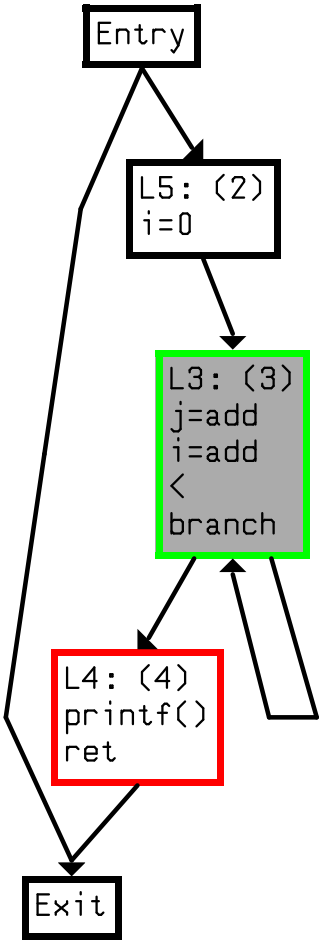
This C is translated back from “high” SUIF.

```
extern int main();
extern int printf();

extern int main()
  {
    int i;
    int j;

    for (i = 0; i <= 9; i++)
      {
        j = j + i;
      }
    printf("j = %d\n", j);
    return 0;
  }
```

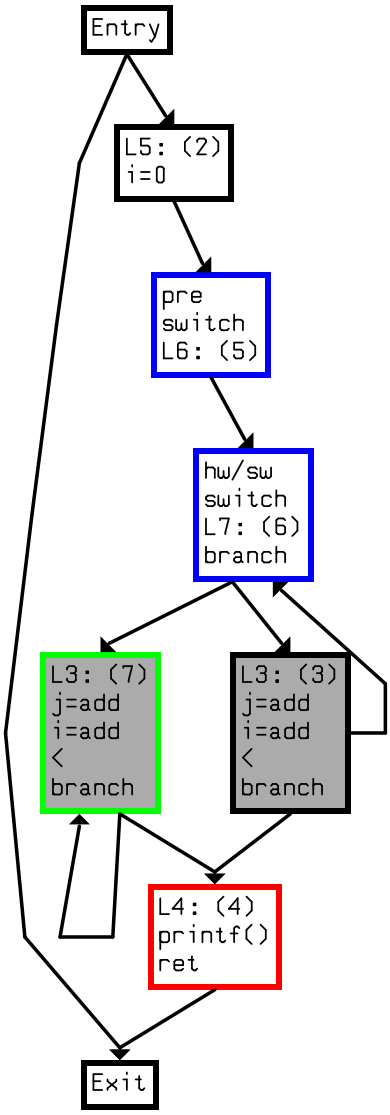
This is a direct translation to C of the optimized and dismantled “low” SUIF.

```
extern int main();
extern int printf();

extern int main()
  {
    int i;
    int j;

    i = 0;
  L3:
    j = j + i;
    i = i + 1;
    if (!(9 < i))
        goto L3;
    printf("j = %d\n", j);
    return 0;
  }
```

This is the control flow graph after initial kernel formation. The block starting with label ‘L3’ is a loop entry and complete loop, and is selected as the kernel.



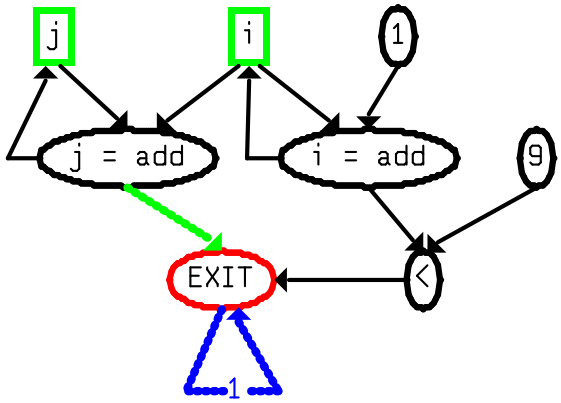
This is the control flow graph after duplication of the initial kernel. The basic block with a faint border is the hardware kernel.



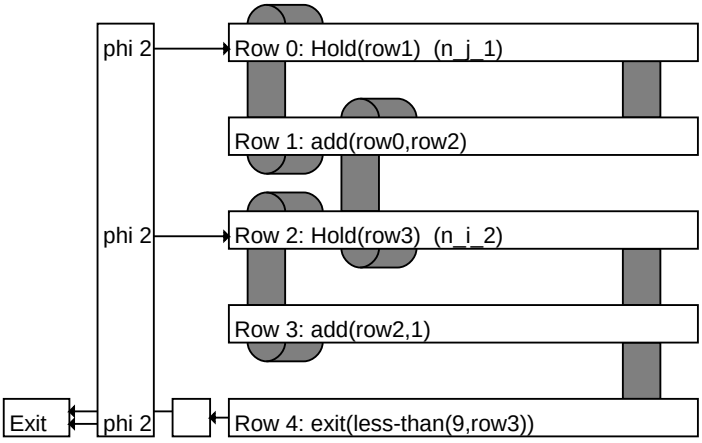
The translation of SUIF back to C with annotations. This is the input to the patching step, which also uses data from the GA file.

```
extern int main();
extern int printf();

extern unsigned int garp_config_prog_main_9_b3[];

extern int main()
  {
    int i;
    int j;
    int garp_kernel_prog_main_9_b3;

  L5:
    i = 0;
  L6: /* pre_switch */
  L7: /* switch */ /* garp_kernel: "prog_main_9_b3" */
    if (garp_kernel_prog_main_9_b3)
        goto L8;
  L3: /* loop_entry */ /* garp_kernel: "prog_main_9_b3" */
    j = j + i;
    i = i + 1;
    if (!(9 < i))
        goto L7;
  L4: /* infeasible: "call:printf+ret+" */
    printf("j = %d\n", j);
    return 0;
  L8: /* garp_kernel: "prog_main_9_b3" */
    /* garp_load_config: garp_config_prog_main_9_b3 */
  L9: /* garp_kernel: "prog_main_9_b3" */
    /* garp_mtga: i j */
  L10: /* garp_kernel: "prog_main_9_b3" */
    /* splitter_root */
    /* exit_num: 0 1 "n___7" */
    /* garp_mfga: j "n_j_0" */
    goto L4;
  }
```

This is the DFG constructed for the loop. The variable ‘j’ is live at the exit.



This is an automatically generated, very simplified representation of the synthesized datapath.



This is the GA file—the output from GAMA synthesis. Lines starting with two dashes are comments from the point of view of the final translation step to bits. However the comments containing "symtab" are needed by garpcc to perform proper hardware-software interfacing.

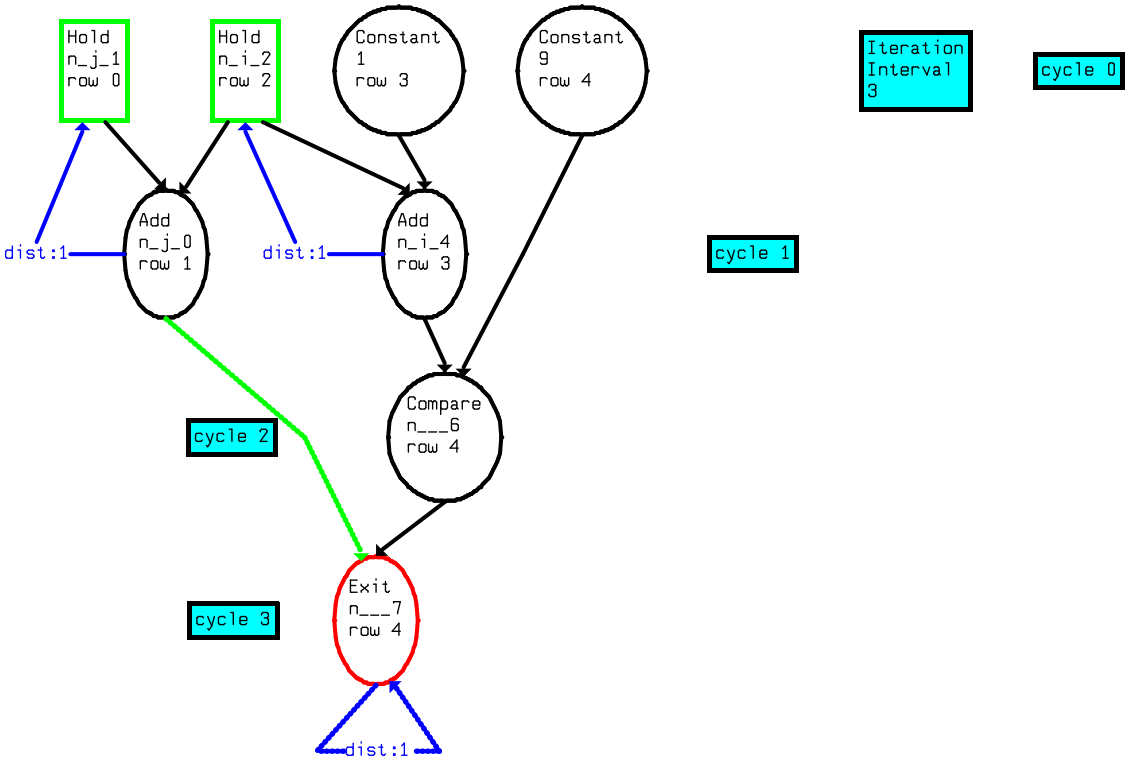
```
-- Gama, Garp version
-- options: do_compress do pre_estimate quick_exit
-- options: CF_DELAY
-- options: size_cutoff:0.900000 z_size_cutoff:0.900000
         far_threshold:6
-- Misc.lib file not found, using default latencies.
-- shrink: looking at Add(4)/n_j_0
-- s not cast
-- shrink: looking at Hold(17)/n_j_1
-- s not cast
-- findLayout: 1 domains.
-- A: 5 modules
-- findLayout: 1 domains.
-- cohab: found 0 bools, 2 dps
-- precedence constraints:
-- row4 must execute 0 or more cycles after row1
-- B: 5 modules
-- Mapping summary: max delay 3 ii 3 utilizing 5 rows with 0 for
         piping
----------------------------
row .row0:
{
-- function Hold(row1) (cycle 2)
     22:   Gout(D,global0);
     --:   trigger cycle 2 (lag 0)
     --:   trigger 2 cycles after row 3 (after row 3)
     22:   A(.row3,hi),functionhi(A),bufferZ,
           D(Zreg),bufferD,Hout(D),Vout(D,long);
-- top: 0x810fd08 attrib: drives_short Attributes(256)
-- p: 0x810fd08 attrib: drives_short Attributes(256)
-- symtab delay_init Zdata n_j_1
   4-19:   A(.row1);
   4-19:    B(global0,hi),C(Zreg),function((B&A)|(~B&C));
   4-19:    bufferZ;
-- symtab output Ddata n_j_1
   4-19:    D(Zreg),bufferD;
-- output from 0x0x810fd08/Hold
   4-19:   Vout(Z);

}
```

```
----------------------------
row .row1:
{
-- function add(row0,row2) (cycle 1)
-- top: 0x810fc60 attrib: Attributes(1)
-- p: 0x810fc60 attrib: Attributes(1)
-- symtab output Zdata n_j_0
   4-19:   A(.row0);
   4-19:   B(.row2);
      4:   shiftzeroin;
   4-19:   bufferZ,U(AˆB), V(A), result(UˆK);
-- output from 0x0x810fc60/Add
   4-19:   Vout(Z,long);

}

----------------------------
row .row2:
{
-- function Hold(row3) (cycle 2)
     22:   Gout(D,global0);
     --:   trigger cycle 2 (lag 0)
     --:   trigger 2 cycles after row 3 (after row 3)
     22:   A(.row3,hi),functionhi(A),bufferZ,
           D(Zreg),bufferD,Hout(D),Vout(D,long);
-- top: 0x810fdb0 attrib: drives_short Attributes(256)
-- p: 0x810fdb0 attrib: drives_short Attributes(256)
-- symtab delay_init Zdata n_i_2
   4-19:   A(.row3);
   4-19:    B(global0,hi),C(Zreg),function((B&A)|(˜B&C));
   4-19:    bufferZ;
-- symtab output Ddata n_i_2
   4-19:    D(Zreg),bufferD;
-- output from 0x0x810fdb0/Hold
   4-19:   Vout(Z);

}

----------------------------
row .row3:
{
-- function add(row2,1) (cycle 1)
-- top: 0x810fe58 attrib: drives_short IsStart Attributes(1)
```

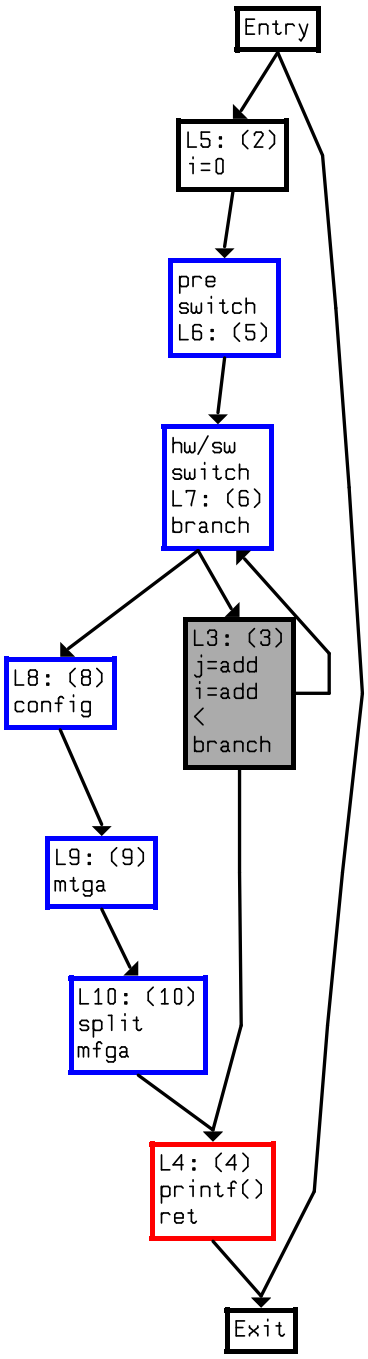
```
-- p: 0x810fe58 attrib: drives_short IsStart Attributes(1)
-- symtab output Zdata n_i_4
   4-19:   A(.row2);
   4-19:   .const_3_B = 0x00000001;
   4-19:   B(.const_3_B);
      4:   shiftzeroin;
   4-19:   bufferZ,U(AˆB), V(A), result(UˆK);
-- Dummy control delay blah
-- symtab start_node 0
     --:   trigger cycle 0 (lag 0)
     --:   trigger 3 cycles after row 3 (after row 3)
     22:   A(Dreg,hi),functionlo(A),bufferZ,
           B(Zreg,lo),functionhi(B),
           D(Zreg),bufferD,Hout(D),Vout(D,long);
-- output from 0x0x810fe58/Add
   4-19:   Vout(Z);

}

----------------------------
row .row4:
{
-- function exit(less-than(9,row3)) (cycle 3)
control:   A(col22,hi),processor,C(col20,lo);
     20:   Hout(Z);
-- symtab control_output Zdata n___7
     --:   trigger cycle 2 (lag 0)
     --:   trigger 2 cycles after row 3 (after row 3)
     22:   A(.row3,hi),functionhi(A),bufferZ,
           D(Zreg),bufferD,Hout(D),Vout(D,long);
     -- exit condition echo in 21lo
     21:   B(col22,hi),C(col20,lo),functionlo(B&C),bufferZ;
-- top: 0x8110050 attrib: Attributes(6)
-- p: 0x8106ae8 attrib: col20
-- symtab output Zcol20 n___6
   4-19:   .const_4_A = 0x00000009;
   4-19:   A(.const_4_A);
   4-19:   B(.row3);
     19:   C(10);
   4-18:   C(00);
      4:   shiftzeroin;
   4-19:   U(˜AˆB), V(BˆC);
     20:   U(00),V(00),result(KˆU),bufferZ;
```

```
}
```

This DFG representation indicates scheduling of operations by GAMA. Merging into modules can be deduced by finding nodes located in the same row.



This is the control flow graph after insertion of all basic blocks for interfacing software to hardware. Note that the hardware copy of the loop has been removed from the software representation.



This is the patched C program ready for final compilation by the MIPS/Garp `gcc`. Information from the GA file has been used to determine the correct array destination/source for MTGA (move to garp array) and MFGA (move from garp array) instrcutions. For example, in basic block "`L9:`", variable '`j`' is moved to the Z register of row 0 of the configuration, which is the Hold module for '`j`'.

```
extern int main();
extern int printf();

extern unsigned int garp_config_prog_main_9_b3[];

extern int main()
  {
    int i;
    int j;
    int garp_kernel_prog_main_9_b3;

  L5:
    i = 0;
  L6: /* pre_switch */
  L7: /* switch */ /* garp_kernel: "prog_main_9_b3" */
        goto L8;
  L3: /* loop_entry */ /* garp_kernel: "prog_main_9_b3" */
    j = j + i;
    i = i + 1;
    if (!(9 < i))
        goto L7;
  L4: /* infeasible: "call:printf+ret+" */
    printf("j = %d\n", j);
    return 0;
  L8: /* garp_kernel: "prog_main_9_b3" */
    /* garp_load_config: garp_config_prog_main_9_b3 */
    asm volatile("gaconf  %0"  :: "r"
         (garp_config_prog_main_9_b3));
  L9: /* garp_kernel: "prog_main_9_b3" */
    /* garp_mtga: i j */
    asm volatile("mtgavz %0,%1" :: "r" (8192),  "r" (7));
    asm volatile("mtga %0,$z2" :: "r" (i));
    asm volatile("mtga %0,$z0" :: "r" (j));
    asm volatile("gabump %0"   :: "r" (0x80000000));
  L10: /* garp_kernel: "prog_main_9_b3" */
    /* splitter_root */
    /* exit_num: 0 1 "n___7" */
```

```
    /* garp_mfga: j "n_j_0" */
    asm volatile("mfga %0,$z1" :  "=r" (j));
    goto L4;
  }

#include "prog.kernels.c"
```

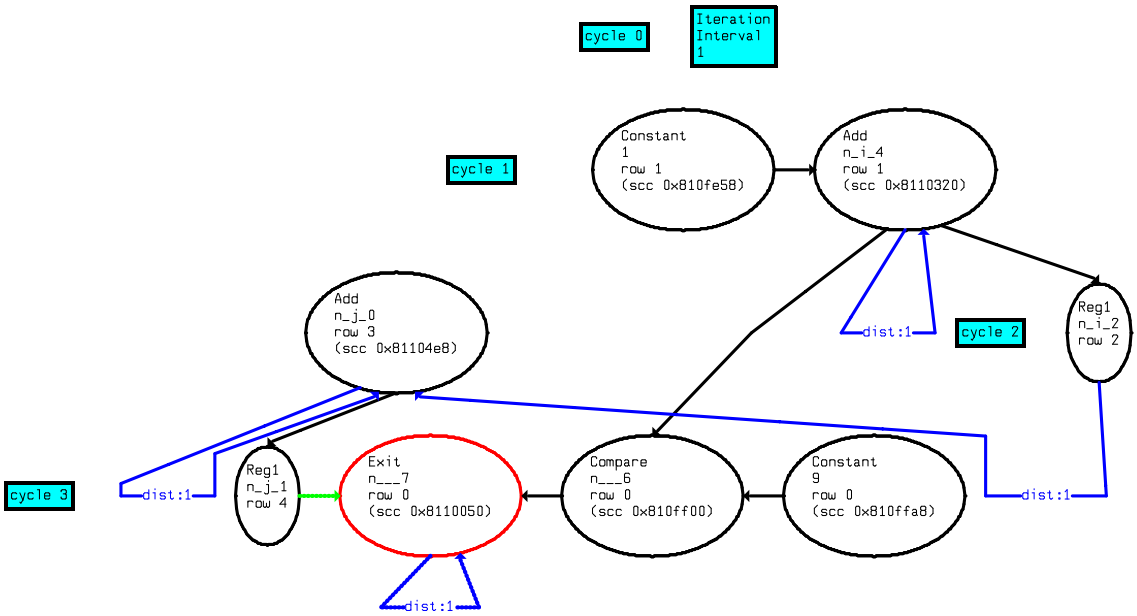
The configuration expressed as array initialization data. This is linked with the final Garp executable. The first word indicates that this is a 5-row configuration. This is followed by 48 32-bit words (192 bytes) of configuration data for each row.

```
{
0x00000005,
0x00000000, 0x00000008, 0x6B000009, 0xAA003FE0, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800,
0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800,
0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800,
0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800,
0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800,
0x7AFF0A0A, 0xB8B81800, 0x7AFF0A0A, 0xB8B81800, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000008, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AAB01F,
0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AAB01F,
0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AAB01F,
0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AAB01F,
0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AAB01F,
0x967A0002, 0x66AAB01F, 0x967A0002, 0x66AA901F, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000008, 0x6B000009, 0xAA003FE0, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E,
0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E,
0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E,
0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E,
0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E,
0x7EFF0A0A, 0xB8B8181E, 0x7EFF0A0A, 0xB8B8181E, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000008, 0x0F080009, 0xCCAA3D1A, 0x00000000, 0x00000000,
```

0x00000000, 0x00000000, 0x7E020002, 0x66AAB01E, 0x7E020002, 0x66AAB01E,
0x7E020002, 0x66AAB01E, 0x7E020002, 0x66AAB01E, 0x7E020002, 0x66AAB01E,
0x7E020002, 0x66AAB01E, 0x7E020002, 0x66AAB01E, 0x7E020002, 0x66AAB01E,
0x7E020002, 0x66AAB01E, 0x7E020002, 0x66AAB01E, 0x7E020002, 0x66AAB01E,
0x7E020002, 0x66AAB01E, 0x7E020002, 0x66AAB01E, 0x7E020002, 0x66AAB01E,
0x7E020002, 0x66AAB01E, 0x7E050002, 0x66AA901E, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0xDB00E000, 0x0000000A, 0x6F000009, 0xAA003D00, 0x00D3D801, 0x00C03000,
0x00000002, 0x0000B000, 0x027A0600, 0x993CA000, 0x027A0200, 0x993CA000,
0x027A0200, 0x993CA000, 0x027A0200, 0x993CA000, 0x027A0200, 0x993CA000,
0x027A0200, 0x993CA000, 0x027A0200, 0x993CA000, 0x027A0200, 0x993CA000,
0x027A0200, 0x993CA000, 0x027A0200, 0x993CA000, 0x027A0200, 0x993CA000,
0x027A0200, 0x993CA000, 0x027A0200, 0x993CA000, 0x027A0200, 0x993CA000,
0x067A0200, 0x993CA000, 0x057A0200, 0x993C8000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
}

A.2 Pipelined example

The same program as the previous section is recompiled with pipelining turned on. There is no difference until the synthesis and patching stages. Hold nodes are eliminated; registers are inserted to ensure data arrives at module inputs synchronized for the same iteration.



This shows the GA file. Note the `prolog_init` indicators in the comments.

```
-- Gama, Garp version
-- options: do_compress do pre_estimate quick_exit
-- options: CF_DELAY
-- options: size_cutoff:0.900000 z_size_cutoff:0.900000
        far_threshold:6
-- Misc.lib file not found, using default latencies.
-- shrink: looking at Add(4)/n_j_0
-- s not cast
-- shrink: looking at Hold(17)/n_j_1
-- s not cast
-- findLayout: 1 domains.
-- A: 3 modules
-- findLayout: 1 domains.
-- precedence constraints:
-- row0 must execute 0 or more cycles after row2
-- B: 3 modules
-- findLayout: 1 domains.
-- Mapping summary: max delay 3 ii 1 utilizing 5 rows with 2 for
        piping
----------------------------
row .row0:
{
-- function exit(less-than(9,row1)) (cycle 3)
control:   A(col22,hi),processor,C(col20,lo);
     20:   Hout(Z);
-- symtab control_output Zdata n___7
-- symtab live_var n___7 n_j_0 4
     --:   trigger cycle 2 (lag 0)
     --:   trigger 2 cycles after row 4 (after row 4)
     22:   A(.row4,hi),functionhi(A),bufferZ,
           D(Zreg),bufferD,Hout(D),Vout(D,long);
     -- exit condition echo in 21lo
     21:   B(col22,hi),C(col20,lo),functionlo(B&C),bufferZ;
-- top: 0x8110050 attrib: Attributes(6)
-- p: 0x810ff00 attrib: col20
   4-19:   .const_0_A = 0x00000009;
   4-19:   A(.const_0_A);
   4-19:   B(.row1);
     19:   C(10);
   4-18:   C(00);
      4:   shiftzeroin;
   4-19:   U(~A^B), V(B^C);
     20:   U(00),V(00),result(K^U),bufferZ;
```

```
}

----------------------------
row .row1:
{
-- function add(row1,1) (cycle 1)
-- top: 0x810fdb0 attrib: drives_short new_prolog_init(0)
         new_prolog_sym(n_i_2) Attributes(1)
-- p: 0x810fdb0 attrib: drives_short new_prolog_init(0)
         new_prolog_sym(n_i_2) Attributes(1)
   4-19:   A(Zreg);
   4-19:   .const_1_B = 0x00000001;
   4-19:   B(.const_1_B);
      4:   shiftzeroin;
   4-19:   bufferZ,U(AˆB), V(A), result(UˆK);
-- output from 0x0x810fdb0/Add
   4-19:   Vout(Z);
-- symtab prolog_init Zdata n_i_2 0

}

----------------------------
row .row2:
{
-- function Reg1(row1) (cycle 2)
-- top: 0x8114c08 attrib: Attributes(16384)
-- p: 0x8114c08 attrib: Attributes(16384)
   4-19:   A(.row1);
   4-19:    function(A),bufferZ;
-- output from 0x0x8114c08/Reg1
   4-19:   Vout(Z,long);

}

----------------------------
row .row3:
{
-- function add(row3,row2) (cycle 2)
-- top: 0x810fc60 attrib: drives_short new_prolog_init(1)
         new_prolog_sym(n_j_1) Attributes(1)
-- p: 0x810fc60 attrib: drives_short new_prolog_init(1)
         new_prolog_sym(n_j_1) Attributes(1)
   4-19:   A(Zreg);
```

```
   4-19:   B(.row2);
      4:   shiftzeroin;
   4-19:   bufferZ,U(AˆB), V(A), result(UˆK);
-- output from 0x0x810fc60/Add
   4-19:   Vout(Z);
-- symtab prolog_init Zdata n_j_1 1

}

----------------------------
row .row4:
{
-- function Reg1(row3) (cycle 3)
-- top: 0x8114da0 attrib: drives_short IsStart Attributes(16384)
-- p: 0x8114da0 attrib: drives_short IsStart Attributes(16384)
   4-19:   A(.row3);
   4-19:    function(A),bufferZ;
-- symtab pipe_output Zdata n_j_0 n___7
-- symtab pipe_output Zdata n_j_0 n___7
-- Dummy control delay blah
-- symtab start_node 0
     --:   trigger cycle 0 (lag 0)
     --:   trigger 1 cycles after row 4 (after row 4)
     22:   D(Dreg),bufferD,Hout(D),Vout(D,long);
-- output from 0x0x8114da0/Reg1
   4-19:   Vout(Z);

}
```

This is the patched C file. MTGA instructions are still used for moving data to the array, but they may occur at different cycles. The optional third operand to an MTGA instruction is a small integer causing the array to advance that many cycles before the next instruction executes.

```
extern int main();
extern int printf();

extern unsigned int garp_config_prog_main_9_b3[];

extern int main()
  {
    int i;
    int j;
    int garp_kernel_prog_main_9_b3;

  L5:
    i = 0;
  L6: /* pre_switch */
  L7: /* switch */ /* garp_kernel: "prog_main_9_b3" */
        goto L8;
  L3: /* loop_entry */ /* garp_kernel: "prog_main_9_b3" */
    j = j + i;
    i = i + 1;
    if (!(9 < i))
        goto L7;
  L4: /* infeasible: "call:printf+ret+" */
    printf("j = %d\n", j);
    return 0;
  L8: /* garp_kernel: "prog_main_9_b3" */
    /* garp_load_config: garp_config_prog_main_9_b3 */
    asm volatile("gaconf  %0"  :: "r"
         (garp_config_prog_main_9_b3));
  L9: /* garp_kernel: "prog_main_9_b3" */
    /* garp_mtga: i j */
    asm volatile("mtgavz %0,%1" :: "r" (8192),  "r" (9));
    asm volatile("mtga %0,$z1,1" :: "r" (i));
    asm volatile("mtga %0,$z3" :: "r" (j));
    asm volatile("gabump %0"   :: "r" (0x80000000));
  L10: /* garp_kernel: "prog_main_9_b3" */
    /* splitter_root */
    /* exit_num: 0 1 "n___7" */
    /* garp_mfga: j "n_j_0" */
    asm volatile("mfga %0,$z4" :  "=r" (j));
    goto L4;
```

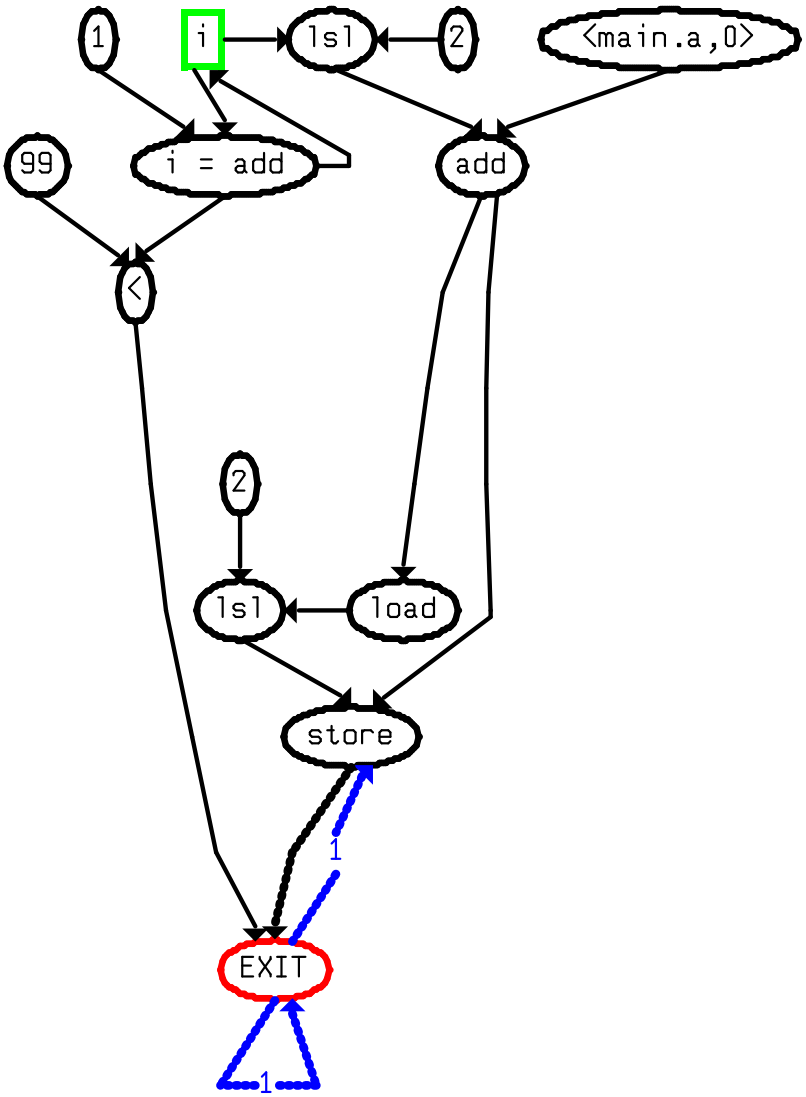
```
  }

#include "prog.kernels.c"
```

A.3 Simple example with memory access

This example accesses memory. In this section, queue recognition is not enabled, so instead “demand” memory accesses are used.

```
main()
{
int i;
int a[100];

        for (i=0; i<100; i++) a[i] = a[i] << 2;
        printf("a[10] = %d\n", a[10]);

}
```

This is the resulting DFG:



The GA file shows how rows involved with memory accesses need to interact with the control blocks.

```
-- Gama, Garp version
-- options: do_compress do pre_estimate quick_exit
-- options: CF_DELAY
-- options: size_cutoff:0.900000 z_size_cutoff:0.900000
         far_threshold:6
-- Misc.lib file not found, using default latencies.
-- shrink: looking at Shift(6)/n___2
-- s not cast
-- shrink: looking at Shift(6)/n___11
-- s not cast
-- checking n___12 for exclusive stores
-- findLayout: 2 domains.
-- A: 7 modules
-- findLayout: 2 domains.
-- check op8 8110ae0 asap 5 ii 7
-- precedence constraints:
-- row0 must execute 1 or more cycles after row6
-- B: 7 modules
-- sourceabs: Load/n___9: 1
-- findLayout: 2 domains.
-- Mapping summary: max delay 8 ii 2 utilizing 13 rows with 6 for
         piping
----------------------------
row .row0:
{
-- function Reg2(row1) (cycle 5)
-- top: 0x811afe0 attrib: Attributes(16384)
-- p: 0x811afe0 attrib: Attributes(16384)
   4-19:   A(.row1);
   4-19:    function(A),bufferZ, D(Zreg), bufferD;
-- output from 0x0x811afe0/Reg2
   4-19:   Vout(D,long);

}

----------------------------
row .row1:
{
-- function Reg1(row2) (cycle 3)
-- top: 0x811ae48 attrib: drives_short Attributes(16384)
-- p: 0x811ae48 attrib: drives_short Attributes(16384)
```

```
   4-19:   A(.row2);
   4-19:    function(A),bufferZ;
-- output from 0x0x811ae48/Reg1
   4-19:   Vout(Z);

}

----------------------------
row .row2:
{
-- function add(row1,1) (cycle 2)
-- top: 0x81106f0 attrib: drives_horiz gets_shifted_left
         drives_short new_prolog_init(0) new_prolog_sym(n_i_3)
         Attributes(1)
-- p: 0x81106f0 attrib: drives_horiz gets_shifted_left
         drives_short new_prolog_init(0) new_prolog_sym(n_i_3)
         Attributes(1)
   4-19:   A(.row1);
   4-19:   .const_2_B = 0x00000001;
   4-19:   B(.const_2_B);
      4:   shiftzeroin;
   4-19:   bufferZ,U(AˆB), V(A), result(UˆK);
-- output from 0x0x81106f0/Add
   4-19:   Vout(Z),Hout(Z);
control:   Hleft;
-- symtab prolog_init Zdata n_i_3 0

}

----------------------------
row .row3:
{
-- function add(Patch:a,shift_left(row2,2)) (cycle 1)
-- top: 0x8110990 attrib: Attributes(8193)
-- p: 0x8110990 attrib: Attributes(8193)
-- found a constant shift
-- found an even shift
-- found a constant shift
-- found an even shift
-- symtab patch Ddata a + 0
   4-19:   A(Dreg),D(Dreg),bufferD; -- input a
   4- 4:   B(00);
   5-19:   B(aboveright1);
      4:   shiftzeroin;
```

```
   4-19:   bufferZ,U(A^B), V(A), result(U^K);
-- output from 0x0x8110990/Add
   4-19:   Vout(Z,long);

}

----------------------------
row .row4:
{
-- function addr(Zlatch(row3)) (cycle 2)
control:   memory,demand(size32*1),latency(3),
           A(col22,hi),B(1),C(0),D(0);
     --:   trigger cycle 1 (lag 0)
     --:   trigger 1 cycles after row 9 (after row 9)
     22:   D(.row9),bufferD,Hout(D),Vout(D,long);
-- top: 0x8110b88 attrib: Attributes(8)
-- p: 0x8111518 attrib:
-- ga misc
   4-19:   A(.row3);
   4-19:   bufferZ,function(A);

}

----------------------------
row .row5:
{
-- function Reg2(row3) (cycle 3)
-- top: 0x811b250 attrib: Attributes(16384)
-- p: 0x811b250 attrib: Attributes(16384)
   4-19:   A(.row3);
   4-19:    function(A),bufferZ, D(Zreg), bufferD;
-- output from 0x0x811b250/Reg2
   4-19:   Vout(D,long);

}

----------------------------
row .row6:
{
-- function Reg1(row0) (cycle 6)
-- top: 0x811b0b8 attrib: drives_short Attributes(16384)
-- p: 0x811b0b8 attrib: drives_short Attributes(16384)
   4-19:   A(.row0);
   4-19:    function(A),bufferZ;
```

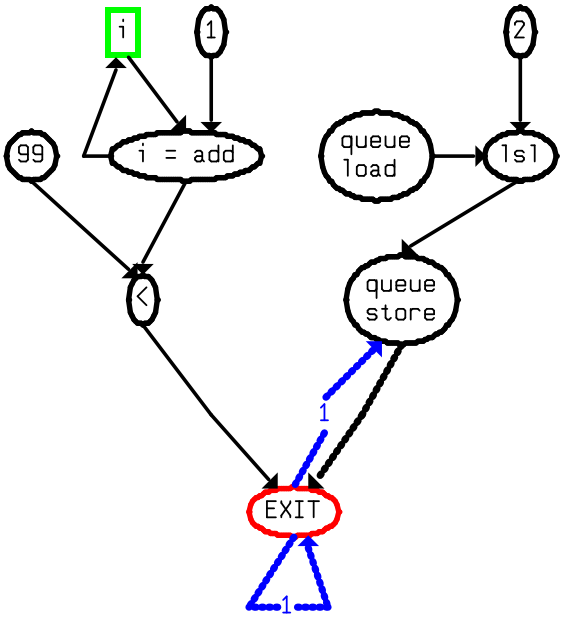
```
-- output from 0x0x811b0b8/Reg1
   4-19:   Vout(Z);

}

----------------------------
row .row7:
{
-- function exit(less-than(99,row6)) (cycle 8)
control:   A(col22,hi),processor,C(col20,lo);
     20:   Hout(Z);
-- symtab control_output Zdata n___17
     --:   trigger cycle 7 (lag 0)
     --:   trigger 6 cycles after row 4 (after row 9)
     22:   A(.row4,hi),functionlo(A),bufferZ,
           D(Zreg),bufferD,Hout(D),Vout(D,long),
           B(col21),functionhi(B);
     21:   A(col22,lo),functionhi(A),bufferZ,
           D(Zreg),bufferD,Hout(D);
     -- exit condition echo in 21lo
     21:   B(col22,hi),C(col20,lo),functionlo(B&C);
-- top: 0x8111020 attrib: Attributes(6)
-- p: 0x8110ed0 attrib: col20
   4-19:   .const_7_A = 0x00000063;
   4-19:   A(.const_7_A);
   4-19:   B(.row6);
     19:   C(10);
   4-18:   C(00);
      4:   shiftzeroin;
   4-19:   U(~A^B), V(B^C);
     20:   U(00),V(00),result(K^U),bufferZ;

}

----------------------------
row .row8:
{
-- function Reg2(row5) (cycle 5)
-- top: 0x811b2f8 attrib: Attributes(16384)
-- p: 0x811b2f8 attrib: Attributes(16384)
   4-19:   A(.row5);
   4-19:    function(A),bufferZ, D(Zreg), bufferD;
-- output from 0x0x811b2f8/Reg2
   4-19:   Vout(D,long);
```

```
}

----------------------------
row .row9:
{
-- function Reg1(row8) (cycle 6)
-- top: 0x811b3b8 attrib: IsStart Attributes(16384)
-- p: 0x811b3b8 attrib: IsStart Attributes(16384)
   4-19:   A(.row8);
   4-19:    function(A),bufferZ;
-- Dummy control delay blah
-- symtab start_node 0
     --:   trigger cycle 0 (lag 0)
     --:   trigger 2 cycles after row 9 (after row 9)
     22:   A(Dreg,hi),functionhi(A),bufferZ,
           D(Zreg),bufferD,Hout(D),Vout(D,long);
-- output from 0x0x811b3b8/Reg1
   4-19:   Vout(Z,long);

}

----------------------------
row .row10:
{
-- function addr(Zlatch(row9)) (cycle 7)
control:   memory,demand(size32*1),
           A(col22,hi),B(1),C(0),D(1);
     --:   trigger cycle 6 (lag 0)
     --:   trigger 2 cycles after row 11 (after row 9)
     22:   A(.row11,hi),functionhi(A),bufferZ,
           D(Zreg),bufferD,Hout(D),Vout(D,long);
-- top: 0x8110e28 attrib: Attributes(8)
-- p: 0x8111a10 attrib:
-- ga misc
   4-19:   A(.row9);
   4-19:   bufferZ,function(A);

}

----------------------------
row .row11:
{
-- function load(row4) (cycle 5)
```

```
-- top: 0x8110ae0 attrib: output_from_d drives_horiz
         gets_shifted_left sourceAbs(1) Attributes(4096)
-- p: 0x8110ae0 attrib: output_from_d drives_horiz
         gets_shifted_left sourceAbs(1) Attributes(4096)
control:   memory,regbus(Dreg,size32,bus0);
control:   A(col22,hi),B(0),C(1),D(0);
     --:   trigger cycle 4 (lag 0)
     --:   trigger 4 cycles after row 9 (after row 9)
     22:   A(.row9,hi),functionlo(A),bufferZ,
           D(Zreg),bufferD,Hout(D),Vout(D,long),
           B(Dreg,lo),functionhi(B);
   4-19:   D(Dreg),bufferD;
-- output from 0x0x8110ae0/Load
   4-19:   Vout(D,long),Hout(D);
control:   Hleft;

}

----------------------------
row .row12:
{
-- function store(row10,shift_left(row11,2)) (cycle 7)
control:   memory,regbus(Zreg,size32,bus0);
control:   A(col22,hi),B(0),C(1),D(1);
     --:   trigger cycle 6 (lag 0)
     --:   trigger 2 cycles after row 11 (after row 9)
     22:   A(.row11,hi),functionhi(A),bufferZ,
           D(Zreg),bufferD,Hout(D),Vout(D,long);
-- top: 0x8110d80 attrib: Attributes(8320)
-- p: 0x8110c30 attrib:
-- found a constant shift
-- found an even shift
-- ga misc
   4- 4:   A(00);
   5-19:   A(aboveright1);
   4-19:   bufferZ,function(A);

}
```

A.4 Queue access example

This is the same example as the previous section, but compiled with queue recognition enabled. This is the DFG with the two accesses converted to queues:



The final patched C code must also set up the queue control block for each queue and then install it with a `galqc` instruction. Also, after kernel execution, a `gareset` instruction is required to flush the store queue.

```
extern int main();
extern int printf();

extern unsigned int garp_config_prog_main_9_b3[];

extern int main()
  {
    int i;
    int a[100];
    int garp_kernel_prog_main_9_b3;
    static unsigned int _garp_queue_9[5] =
      {
        16777472u, 33554432u, 0u, 0u, 16777216u
      } ;
    static unsigned int _garp_queue_12[5] =
      {
        16843008u, 33554432u, 0u, 0u, 33554432u
      } ;

  L5:
    i = 0;
  L6: /* pre_switch */
  L7: /* switch */ /* garp_kernel: "prog_main_9_b3" */
        goto L8;
  L3: /* loop_entry */ /* garp_kernel: "prog_main_9_b3" */
    a[i] =  2u;
    i = i + 1;
    if (!(99 < i))
        goto L7;
  L4: /* infeasible: "call:printf+ret+" */
    printf("a[10] = %d\n", a[10]);
    return 0;
  L8: /* garp_kernel: "prog_main_9_b3" */
    /* garp_load_config: garp_config_prog_main_9_b3 */
    asm volatile("gaconf  %0"  :: "r"
         (garp_config_prog_main_9_b3));
  L9: /* garp_kernel: "prog_main_9_b3" */
    /* queue_init: "store" "n___12" _garp_queue_12 32
         "(((char*)a)+((i)*4))" */
    _garp_queue_12[2] = (unsigned)(((char*)a)+((i)*4));
```

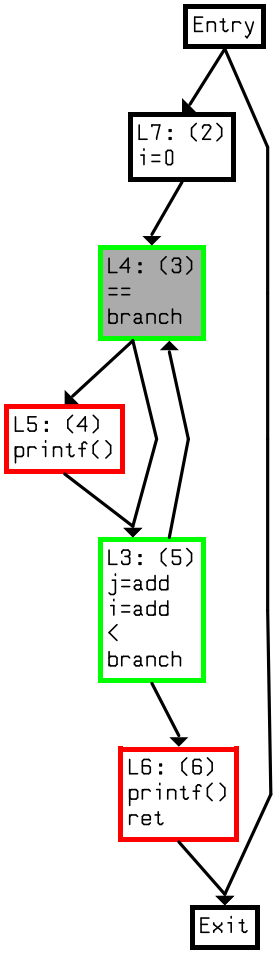
```
    asm volatile("galqc %0,%1" :: "r" (_garp_queue_12), "r" (1));
    /* queue_init: "load" "n___9" _garp_queue_9 32
         "(((char*)a)+((i)*4))" */
    _garp_queue_9[2] = (unsigned)(((char*)a)+((i)*4));
    asm volatile("galqc %0,%1" :: "r" (_garp_queue_9), "r" (0));
    /* garp_mtga: i */
    asm volatile("mtgavz %0,%1" :: "r" (8192),  "r" (1));
    asm volatile("gabump %0"   :: "r" (2));
    asm volatile("mtga %0,$z0" :: "r" (i));
    asm volatile("gabump %0"   :: "r" (0x80000000));
  L10: /* garp_kernel: "prog_main_9_b3" */
    /* splitter_root */
    /* exit_num: 0 1 "n___17" */
    /* garp_mfga */
    asm volatile("mfga $0,$z0");
    asm volatile("gareset");
    goto L4;
  }

#include "prog.kernels.c"
```

A.5 Example with multiple exits

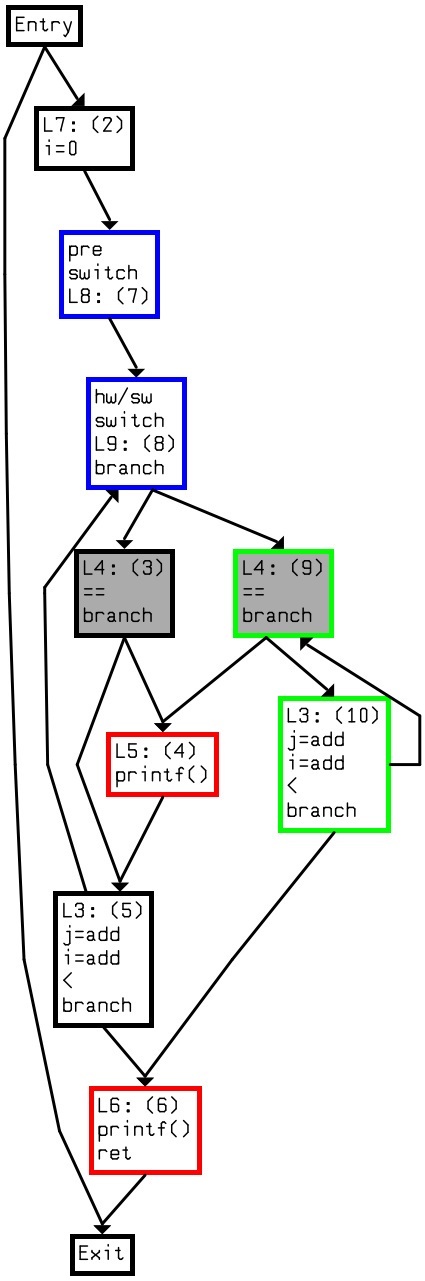
This simple example has two exits. In this case the second exit is caused by the infeasible `printf()` statement.

```
main()
{
int i, j;

        for (i=0; i<10; i++) {
            if (j == 44) printf("hit 44\n");
            j += i;
        }
        printf("j = %d\n", j);
}
```

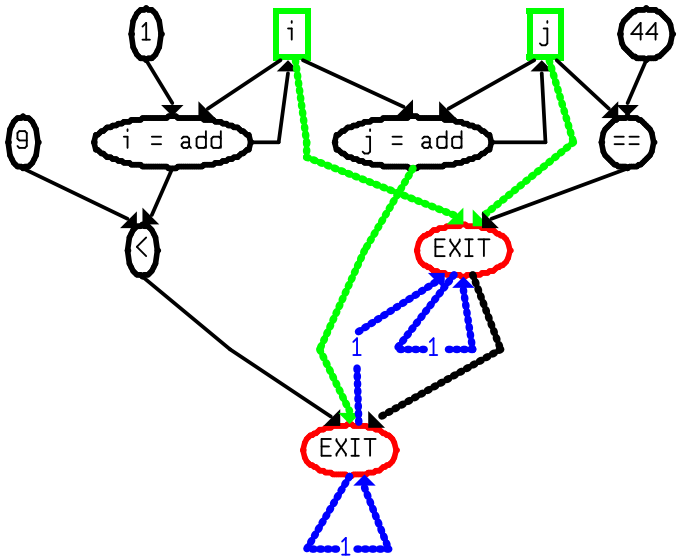
Here is the original CFG:



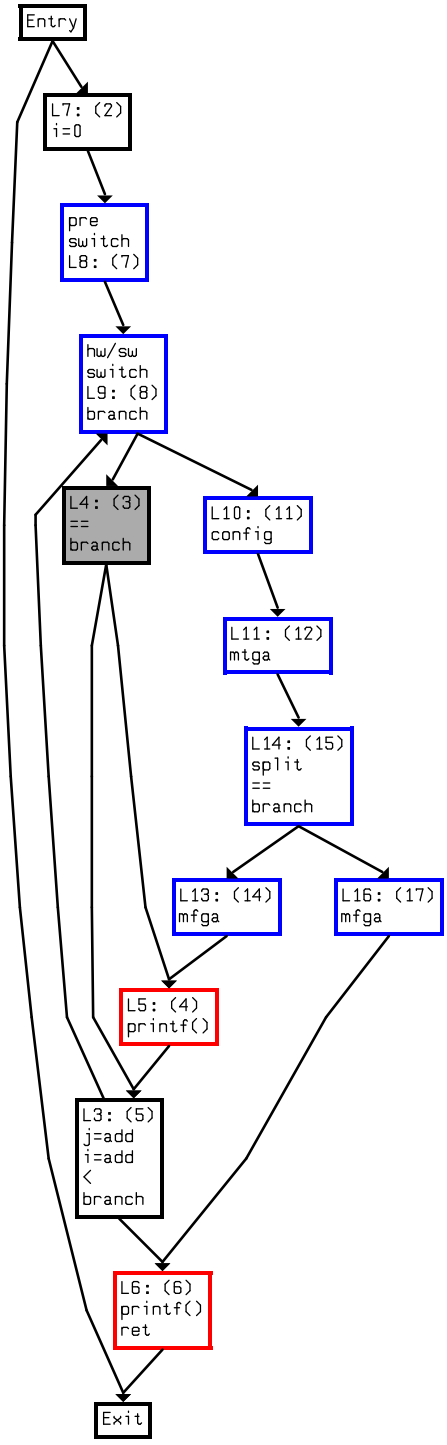
Here is the CFG after kernel duplication:



Here is the DFG; note that the exit nodes have different liveness edges reaching them:



Here is the CFG after insertion of interfacing instructions:



The final patched C code requires a check of the exit indicator bit from one of the exit rows in order to determine which exit was taken and which subsequent code to branch to. The code at L16 is for the final loop exit; only variable ‘`j`’ is live there. The code at L13 is for the `printf()` exit; both ‘`i`’ and ‘`j`’ are live there, and in addition, the definition of ‘`j`’ is different than at the other exit.

```
extern int main();
extern int printf();

extern unsigned int garp_config_prog_main_9_b3[];

extern int main()
  {
    int i;
    int j;
    int garp_kernel_prog_main_9_b3;
    int garp_splitter_prog_main_9_b3;

  L7:
    i = 0;
  L8: /* pre_switch */
  L9: /* switch */ /* garp_kernel: "prog_main_9_b3" */
        goto L10;
  L4: /* loop_entry */ /* garp_kernel: "prog_main_9_b3" */
    if (!(j == 44))
        goto L3;
  L5: /* infeasible: "call:printf+" */
    printf("hit 44\n") ;
  L3:
    j = j + i;
    i = i + 1;
    if (!(9 < i))
        goto L9;
  L6: /* infeasible: "call:printf+ret+" */
    printf("j = %d\n", j);
    return 0;
  L10: /* garp_kernel: "prog_main_9_b3" */
    /* garp_load_config: garp_config_prog_main_9_b3 */
    asm volatile("gaconf  %0"  :: "r"
         (garp_config_prog_main_9_b3));
  L11: /* garp_kernel: "prog_main_9_b3" */
    /* garp_mtga: i j */
    asm volatile("mtgavz %0,%1" :: "r" (8192),  "r" (15));
```

```
    asm volatile("mtga %0,$z2,1" :: "r" (i));
    asm volatile("mtga %0,$z4" :: "r" (j));
    asm volatile("gabump %0"   :: "r" (0x80000000));
  L14: /* garp_kernel: "prog_main_9_b3" */
    /* splitter_root */
    asm volatile("mfgavz %0,%1" : "=r"
         (garp_splitter_prog_main_9_b3) :  "r" (0));
    if ((garp_splitter_prog_main_9_b3 & (1024)) == (1024))
        goto L13;
  L16: /* garp_kernel: "prog_main_9_b3" */
    /* exit_num: 1 2 "n___10" */
    /* garp_mfga: j "n_j_4" */
    asm volatile("mfga %0,$z5" :  "=r" (j));
    goto L6;
  L13: /* garp_kernel: "prog_main_9_b3" */
    /* exit_num: 0 2 "n___3" */
    /* garp_mfga: i "n_i_5" j "n_j_2" */
    asm volatile("mfga %0,$d7" :  "=r" (i));
    asm volatile("mfga %0,$z6" :  "=r" (j));
    goto L5;
  }

#include "prog.kernels.c"
```