A Compiler for Application-Specific Signal Processors

Kenneth Edward Rimey

September 15, 1989

This research was funded by DARPA contract number N00039-87-C-0182.

Acknowledgements

This work took place in the context of a collaboration between the research groups of Robert Brodersen and my advisor, Paul Hilfinger. Edward Wang and I set out to provide Brodersen's signal-processing and integrated-circuit research group with a compiler for Silage, an unusual programming language designed by Hilfinger. Edward Wang worked on the front end and I worked on the code generator. The intermediate language evolved into RL, and the code generator evolved into the compiler described in this thesis.

Many members and associates of Brodersen's group—Lars Svensson, Lars Thon, Chuen-Shen Shung, Markus Thaler, Brian Richards, and Syed Khalid Azim—used the RL compiler in its various stages of development. Lars Thon and his predecessor, Chuen-Shen Shung, maintained the Lager system's interface to the compiler. Lars Svensson, in continuing discussions, has provided insights into the direction in which this research should go in the future.

Paul Hilfinger, Jan Rabaey, and David Auslander served on the thesis committee. Edward Wang, Luigi Semenzato, Kinson Ho, Benjamin Zorn, Lars Svensson, and Lars Thon provided additional helpful comments on the thesis. Edward Wang and Luigi Semenzato, in particular, spent many hours helping me with the hard parts. James Larus helped with previous papers about this work. The emphasis that all of Paul Hilfinger's students seem to place on good writing is partly due to their advisor's influence.

My friends and family deserve thanks for many things. Special thanks go to William Daly of Dumont High School, who encouraged my interest in computers and even hired me as a programmer.

Contents

List of Figures

Chapter 1

Introduction

The impetus for this research has been a project to design a compiler to generate code for a family of application-specific digital signal processors developed at Berkeley. Application-specific processors are programmable processors that serve as components of application-specific integrated circuits.

The compiler that I designed accepts a C-like language called RL and generates code using a machine description provided by the user. My work has demonstrated the utility of a user-retargetable compiler in selecting an appropriate processor architecture for a given program. The designer begins with some preexisting architecture, compiles the program, and then evaluates changes to the architecture by recompiling the program with a modified machine description and observing the effect on the instruction count. I will describe two cases in which this approach has been used at Berkeley to develop signal-processing chips.

The RL compiler's target processors have irregular datapaths like those of typical off-the-shelf signal processors, but execute open horizontal microcode (i.e., the machine instructions are vectors of control signals with little restrictive encoding). The usual technique for generating horizontal microcode is to first generate vertical code and then compact this in a separate pass. This approach is inappropriate for datapaths such as ours, for which I have developed an effective, new technique. It generates and schedules code in a single pass over a straight-line program segment, doing local register allocation and chaining of operations on the fly, using a network-flow algorithm to enforce the constraints that are needed, in this approach, to avoid blocking.

1.1 Application-specific signal processors

The RL compiler was constructed primarily for use with the Lager computer-aided design system [39,31]. Lager provides rapid prototyping of VLSI chips for embedded, real-time applications—especially digital signal processing and control applications.

A way to reduce the design time of a chip is to incorporate a programmable processor.[1]

[1] Lager also addresses very-high-speed applications for which this is not a viable option (e.g., real-time video processing), but these will not be considered here.

When a programmable processor is included on an application-specific chip along with a read-only memory containing the program, it becomes possible to customize the processor architecture to suit the program. The result is an *application-specific processor*. Customizing the architecture is a way to try to achieve or surpass the speed of highly optimized, off-the-shelf digital signal processors.

Perhaps more importantly, it is a way to make the processor small. Hardware that is not needed to achieve the speed required by the application can be discarded, freeing space for integrating additional peripheral circuits, or at least reducing the total chip area (and hence the manufacturing cost). The trend in application-specific integrated circuits is to integrate entire systems onto chips. Analog circuits and even transducers can be brought onto the chip nowadays.

To design application-specific processors quickly, one needs a plastic architectural style. In other words, we want an open-ended family of processors from which to choose. Let us assume, at the outset, that the family uses a horizontal instruction word. In other words, the processors execute a machine language that resembles horizontal microcode. I say "resembles" because, strictly speaking, microcode is what makes up the internal program that emulates the virtual machine in a two-level computer. The proper term to describe our machine language is *open horizontal microcode*.

Using a horizontal instruction word eliminates the need to design instruction sets, instruction formats, and instruction decoding hardware. It lets us focus on designing a datapath suited to the given program. Instruction encoding is an optimization that could be performed later to reduce the size of the program memory, after the code has been generated.

The ambitious approach to selecting a datapath is to build a compiler that generates one at the same time that it generates code for it. This is the spirit of the Cathedral-II system [30]. Cathedral-II generates processor architectures in which each input to each functional unit is associated with a register bank. It chooses a set of functional units using a rule-based system that considers both the source program and additional hints provided by the programmer. It then generates code under the tentative assumption that the outputs of the functional units are connected to the register banks through a full crossbar switch; this fictional target architecture is illustrated in Figure 1.1. After code generation and scheduling, the Cathedral-II system tries to reduce the number of busses in the switch by merging those that are never used simultaneously (or seldom used simultaneously—in this case the scheduling must be redone). It tries to put more than one bus into the same track (creating a split bus) when the order of the required taps permits. It also does register allocation at that time, determining the sizes of the register banks. Figure 1.2 illustrates a plausible result.

The approach that I will describe in this thesis is less automated. The designer selects an interconnection topology (and may size register banks) *before* code generation. Pass-through modes of functional units will augment the interconnect. There may be several register banks, or just one. The compiler will have to be able to generate code for an essentially arbitrary, *irregular* datapath.

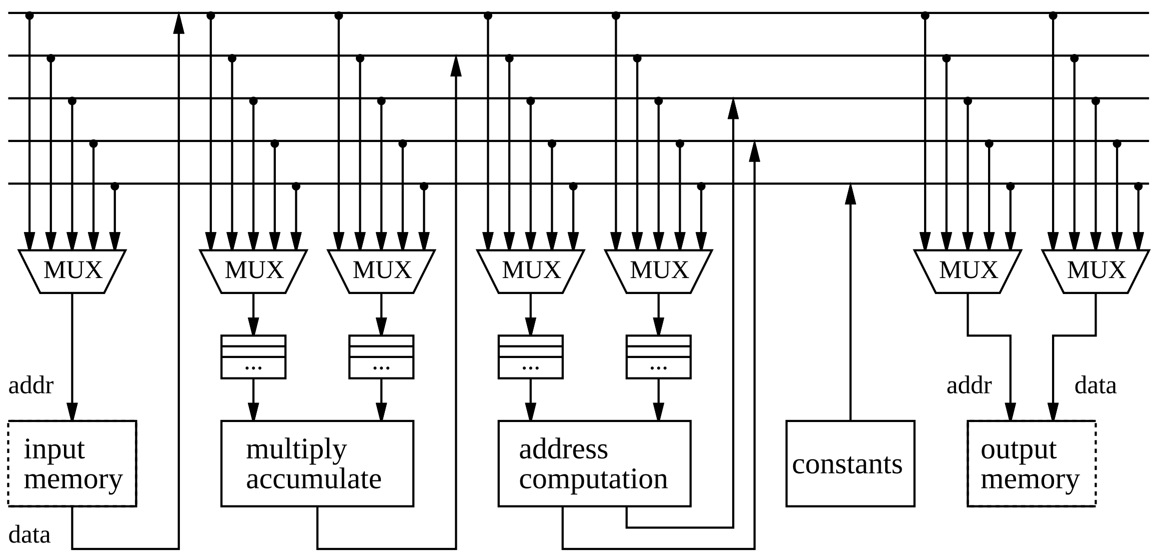

Figure 1.1: An alternative approach: the virtual architecture of Cathedral-II. Shown is a processor with just two execution units, a multiply-accumulate unit and an address-arithmetic unit.

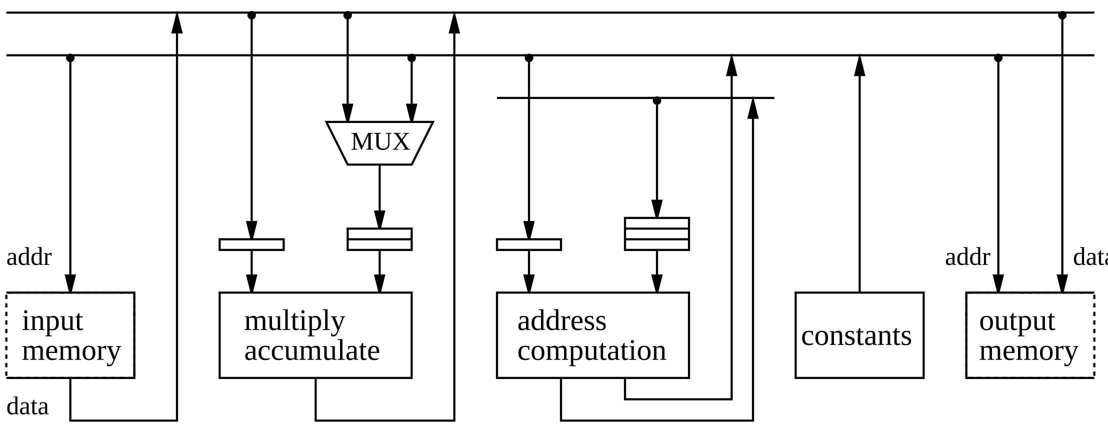

Figure 1.2: The same processor, as Cathedral-II might optimize it after scheduling the code.

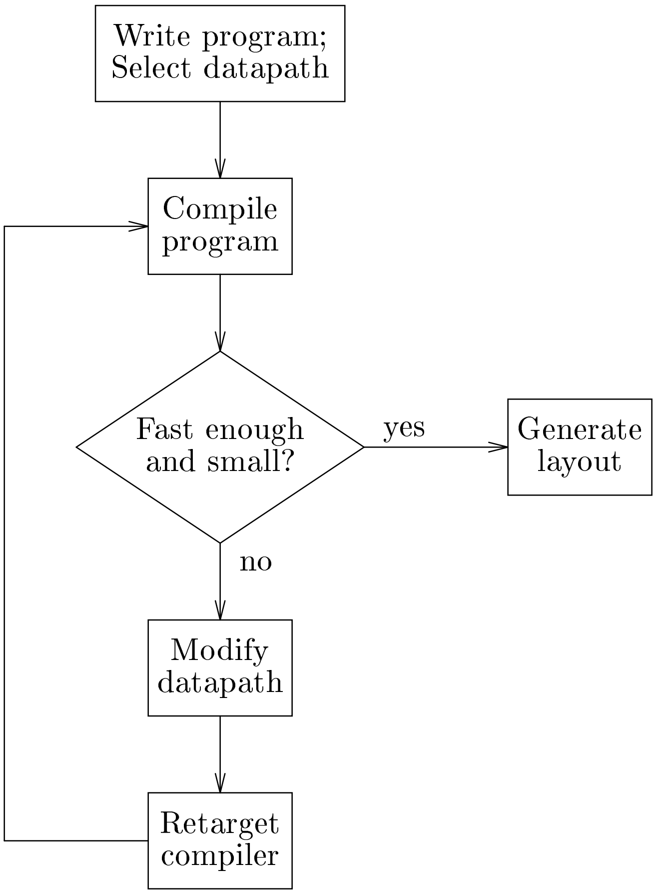

Figure 1.3: The application-specific processor design cycle.

In the fully developed version of this approach, the compiler is easy enough for the chip designer to retarget that he can use it to help design the datapath. He evaluates a change in the design by recompiling the program and observing the change in the instruction count. It may suffice to look at the static instruction count for the whole program, or for an inner loop, or it may be necessary to use simulation data to compute a dynamic instruction count. The overall design cycle is shown in Figure 1.3.

This approach offers advantages over the alternatives, provided that the compiler can handle irregular architectures. A conventional architectural style based on the idea that all intermediate results will be stored in a large, central, multiported register bank simplifies code generation, but makes the circuit design and the layout difficult unless the number of ports is severely limited. The Cathedral-II style is less constraining, but postponing customization until after code generation risks frivolous allocation of resources by the code generator. Generating code for a prespecified, irregular architecture avoids these problems.

Moreover, the task of generating code for an irregular architecture can be reduced to a manageable level of difficulty by minimizing the use of restrictive instruction encoding.

Although generating fully horizontal code is certainly harder than generating vertical code, it is far easier than generating horizontal code under scheduling constraints imposed by a heavily encoded instruction format.

1.2 Generating horizontal microcode

Historically, research on compilers that generate horizontal code has focused on bigger machines, some of which are very different from our application-specific signal processors.

Microengines are used internally in two-level computers to emulate virtual, complex-instruction-set architectures. The Digital Equipment 11/750 and 11/780, for example, are microengines designed for emulating the VAX architecture.

Special-purpose horizontal-instruction-word processors for applications such as digital signal processing and graphics include the VLSI-based, application-specific target architectures of the RL compiler. In the past, special-purpose horizontal-instruction-word processors have more often been implemented on the board level.

Long-instruction-word (LIW) computers, also called scientific array processors, are auxiliary processors for floating-point number crunching. The Floating Point Systems 164 [8] is an LIW machine, as are the ten individual processors of Warp, an architecture for a linear array of processors from Carnegie-Mellon University [24,5]. I would also classify the numeric processor of the Cydrome Cydra as an LIW machine [32].

Very-long-instruction-word (VLIW) supercomputers are similar to LIW computers, except they have many more functional units. The Multiflow Trace is a commercial VLIW supercomputer [9].

Compilation from a high-level language is not taken for granted for the first three classes above. For microengines, the idea has generally been confined to the research lab. Dewitt [13], Mallett [27], Ma [26], and Marwedel [28] are among those who have constructed experimental high-level-language compilers for microengines. Others have constructed compilers for lower level languages; these languages typically provide register transfer notation with Algol-like control constructs. Davidson [11] has surveyed microprogramming languages and devised a taxonomy in which languages are classified by both level of abstractness and degree of machine independence.

Special-purpose processors have worked out better for the microcode compilation community (represented by the yearly SIGMICRO workshop). Gurd [21], Hopkins et al. [22], and a team at Quantitative Technology Corporation have developed C compilers for these machines. JRS Research Laboratories [37] has developed an Ada microcode compiler as a component of its Integrated Design Automation System product. All of these compilers, except for the early one described by Gurd, are intended to be easy to retarget.

LIW computers are usually expected to support Fortran, but there is still an option of writing critical loops by hand or using a hand-written subroutine library such as the

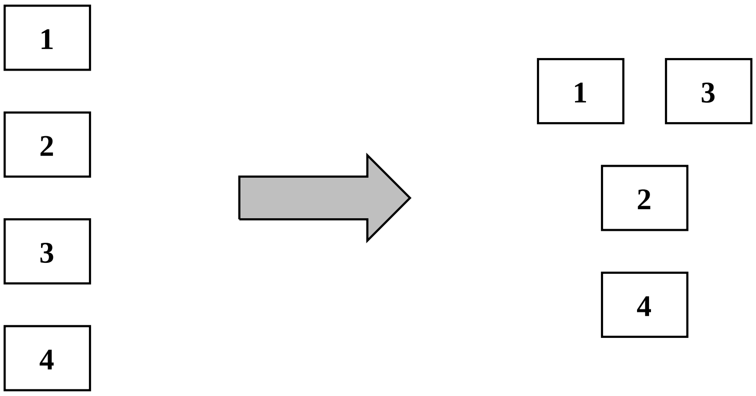

Figure 1.4: Local compaction.

one supplied with the FPS-164. For VLIW supercomputers, a good Fortran compiler is a must because writing code by hand is just too difficult. The Multiflow Trace was, in fact, developed in parallel with its compiler.

Traditional techniques

The usual approach to generating horizontal code has been to separate the process into two phases. First, *vertical* code, the instructions of which usually contain only one microoperation, is generated using techniques borrowed from ordinary compilers. Then the microoperations of the vertical code are packed into a smaller number of instructions in the *compaction* phase. Researchers have studied compaction in various forms:

Local compaction is the packing of straight-line code segments one at a time.

Software pipelining is a specialized technique for the compaction of loops. It rewrites a loop so as to overlap the execution of many successive iterations.

Global compaction generalizes local compaction to include movement of microoperations across branches. The most well-known method is *trace scheduling*, which was developed for VLIW supercomputers. Global compaction does not encompass software pipelining, although it can be used with partial loop unrolling to get almost the same speed-up.

These techniques are illustrated in Figures 1.4 through 1.6. Fisher, Landskov, and Shriver's paper [18] is a good introduction to compaction in general.

Algorithms for local compaction are surveyed by Landskov et al. [25]. Fisher [15] and Davidson et al. [12] conducted experiments to compare them, concluding that simple greedy algorithms with good heuristics perform almost as well as more complex algorithms and even exhaustive search. Since local compaction is now so well-understood, the research community has largely shifted its attention to software pipelining and global compaction.

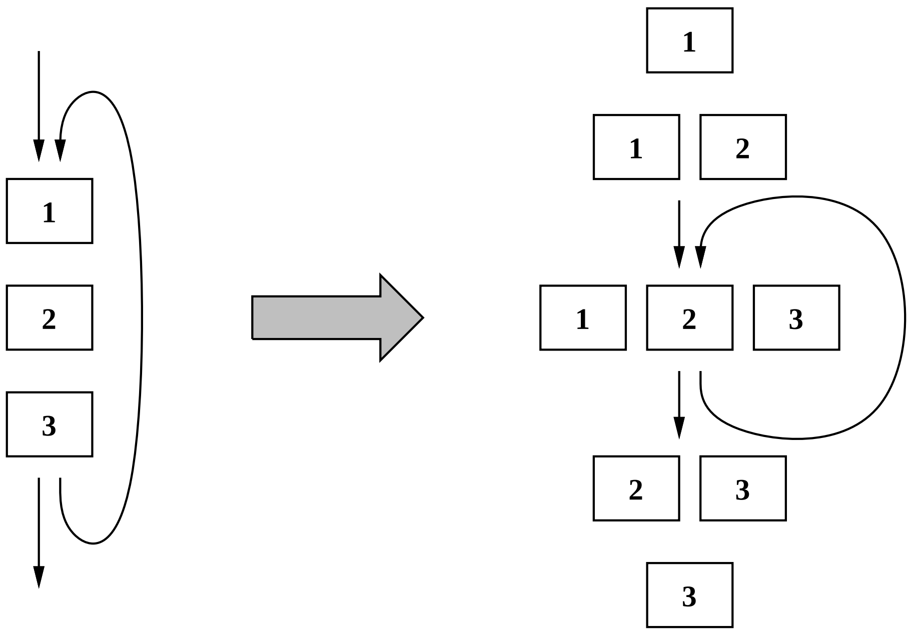

Figure 1.5: Software pipelining.

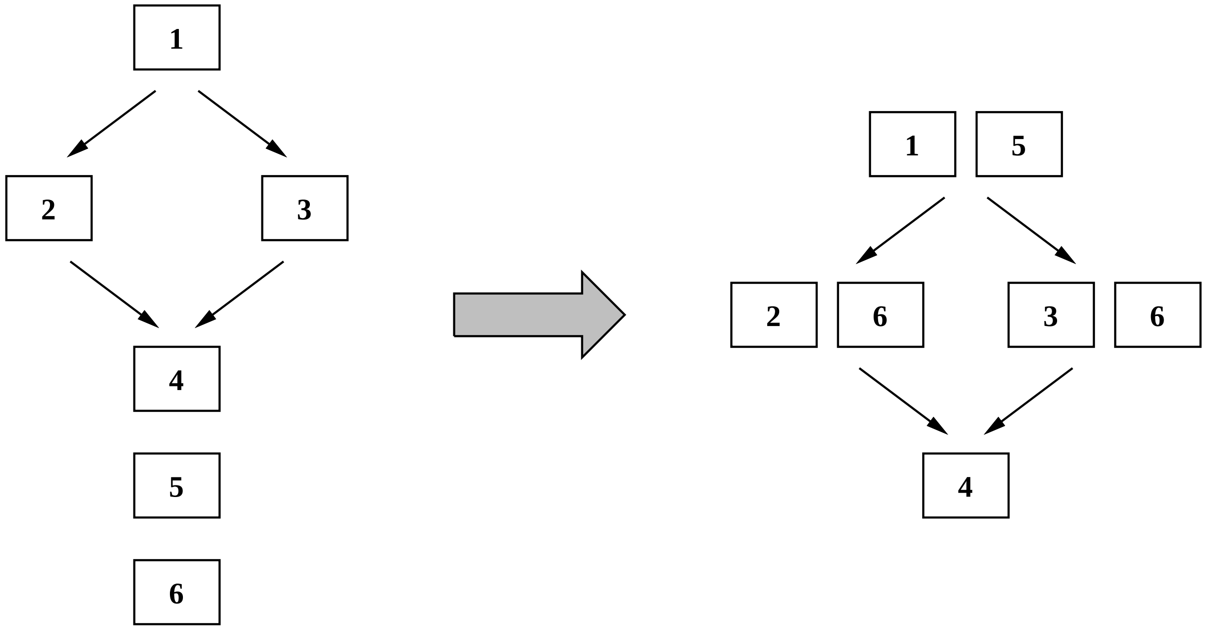

Figure 1.6: Global compaction.

Several authors have addressed software pipelining, proposing widely differing algorithms for the same optimization [45,24,40,2,33]. An LIW computer executing a software-pipelined loop bears some resemblance to a vector processor executing a vector instruction (or several chained vector instructions). This is an attraction of software pipelining: it generalizes vector processing. It also gets by with simpler control hardware.

Touzeau [45] has described a compiler for the FPS-164 that uses software pipelining. On that machine, the fully compacted body of a straightforward inner-product loop (an example used by Charlesworth [8]) is 9 instructions long—i.e., requires 9 cycles to execute. When the loop is software-pipelined, an iteration can be started every other cycle. This is a speed-up of 4.5, which is large indeed. If the loop is rewritten to sum even-numbered and odd-numbered products separately (changing the floating-point result slightly), a speed-up of 9 can be obtained.

Whereas the FPS-164 has one floating-point adder and one floating-point multiplier, the Multiflow Trace has up to four of each. This is a qualitative difference—the Trace can initiate more operations in one cycle than there are altogether in a typical, simple loop. Apparently, partial loop unrolling cannot be avoided for VLIW supercomputers.

One might unroll a little and then apply software pipelining. Or one might unroll more and then just compact the unrolled loop body. As the degree of unrolling is increased, the latter approach produces a compiled loop that approaches the speed of the software-pipelined loop as it takes up more and more space. The result is inferior to the software-pipelined loop, but software pipelining is harder to implement. If the original loop body is straight-line code—let us assume that the loop bounds are known to the compiler—then compaction of the body of the unrolled loop is a local compaction problem. If the original loop contains, say, an `if-then-else` statement, then a global compaction problem results.

Many authors [16,48,29,44,23,1] have proposed algorithms for global compaction. Fisher's *trace scheduling* method is the most well-known. It repeatedly chooses a frequently executed path or *trace* through the control flow graph, compacts it as if it were straight-line code, and then adds fix-up code to edges that enter or leave the trace. Unfortunately, trace scheduling is unsuitable for many signal processing applications because it improves average running time only at the expense of worst-case running time.

Like trace scheduling, Wood's method [48], which is a refinement of an old proposal by Dasgupta [10], reduces global compaction to a number of local compaction problems. It compacts a sequence of basic blocks and structured control constructs such as loops and `if-then-else` statements by first compiling the control constructs and then compacting the whole sequence as if the control constructs were just unusually complicated microoperations. Unlike trace scheduling, it improves worst-case running time and does not depend on estimates of branch frequencies. It happens to do good scheduling of subroutine calls naturally.

Lam, in her dissertation [24], has further refined Wood's technique and given it a name: *hierarchical reduction.* She uses it, in combination with software pipelining, to generate code for the Warp, a small linear array of LIW processors. Data presented in her thesis indicates that the speed-up from hierarchical reduction derives mainly from cases in which it makes possible the software pipelining of loops whose bodies are not straight-line code. Lam's work is the current state of the art in generating LIW code. How her approach compares to trace scheduling in generating VLIW code is still an open question.

Global compaction and software pipelining address the main issue in generating code for LIW and VLIW machines: keeping the pipeline full for each of the floating-point functional units. For some microengines and special-purpose machines, this is not the main issue. The application-specific processors that users of the RL compiler have designed so far do fixed-point, not floating-point, arithmetic and thus rarely incorporate functional units with latencies of more than one instruction cycle. For these machines—assuming that critical, tight loops are partially unrolled in any case—global compaction and software pipelining offer speed-ups of only a few tens of percent. More is to be gained from optimizations that improve the straight-line code.

Vegdahl investigated the generation of straight-line code for microengines [46,47]. He found that it is worthwhile to couple the two phases into which the process is usually divided: feedback from the compaction phase is needed for effective microoperation selection.

His investigation focused on the generation of constants. In a microengine, it is common for there to be several ways to generate the same small integer. For example, there may be several ways to generate the number one besides loading it from memory. To generate the number two, one might load it from memory or one might generate and add together two ones. Which is better will often depend on the opportunities for compaction.

Vegdahl compared several ways of coupling the two phases, but he did not discuss combining them into a unified *scheduler*. The success of greedy, local compaction algorithms suggests a natural approach: Do selection of microoperations on the fly as they are positioned in the final code by a greedy algorithm.

Two different greedy, local scheduling algorithms were developed to compact traces in the trace-scheduling Bulldog VLIW compiler developed by Fisher and his students at Yale [14]. The first, developed by Ellis, is based on list scheduling. It selects functional units in a preliminary pass, rather than on the fly, on the assumption that functional units and not registers or busses are the critical resources in the machine—a reasonable assumption for LIW and VLIW machines. The second, developed by Ruttenberg, is based on what he calls *operation scheduling*. It does select functional units on the fly and it, furthermore, delays all code-generation decisions for as long as it can. Ruttenberg's operation scheduler and Ellis's list scheduler are compared in a 1984 paper written with Fisher and Nicolau: "Parallel Processing: A Smart Compiler and a Dumb Machine" [17].

My approach

The local scheduling algorithm that I will describe in Chapter 4 is also based on operation scheduling and is closely related to the algorithm described by Ruttenberg. It considers the nodes of the dataflow graph for a straight-line code segment in turn, in some order consistent with all data dependences. For each node—or operation—it finds the *function* microoperation that can be inserted into the earliest possible instruction and inserts it. It also finds *transfer* microoperations (which just copy data from place to place) that deliver the arguments of the function microoperation, and inserts them into the appropriate instructions.

There may be many routes along which an argument value can be delivered to the desired place and time from some place and time where it has already been deemed to exist. The task of selecting argument delivery routes is called *data routing*. How best to route an intermediate result between functional units depends on the time interval between the definition and the use, as well as on what resources in the datapath are free during that interval. By postponing the selection of the route until the use is scheduled (and more of the schedule is known), more constraints are obtained to guide the selection. This is *lazy* data routing.

Greedy scheduling of function microoperations with lazy scheduling of transfer microoperations is the essence of operation scheduling. Effective treatment of transfer microoperations seems to be particularly important for our target, application-specific processors.

The hard part of lazy data routing is making sure that all feasible routes for a value are not unwittingly closed off by an unfortunate scheduling decision made between when the value's definition is scheduled and when the last use is scheduled. Ruttenberg's operation scheduler requires that each function microoperation have an associated register bank into which the result is always put. When it schedules a function microoperation, it sets aside a register to hold the result for an indeterminate time.

I propose a more powerful approach in which no fixed register is set aside. Instead, a *spill path*, a route into the indefinite future, is set aside—tentatively. As scheduling proceeds, the spill path is adjusted when necessary. The purpose is just to identify and avoid scheduling decisions for which no accommodating adjustment exists.

The spill-path technique allows the result register to be chosen lazily. Furthermore, it handles the case in which the result is never put into a proper register—just into a series of hot spots: busses, latches, registers that would obstruct computation if tied up. The Bulldog compiler outlawed hot spots altogether.

In Chapter 4, I will describe a network-flow algorithm for incrementally maintaining a set of spill paths. With this algorithm, any number of spill paths can be rerouted to accommodate an incremental change, in a time linear both in the number of elements in the datapath and in the number of instructions in the resulting straight-line code segment.

Chapter 2

Target Architectures

The RL compiler is designed to generate code for a family of application-specific processor architectures, the members of which I will call *Kappa-like* architectures because they are derived for the most part from an architecture called *Kappa*. Although Kappa itself was designed for a particular robot-control application [7,6], it is not a very specialized architecture, and it has served well as a prototype for other application-specific processors developed within the Lager project.

All Kappa-like processors use a horizontal instruction format, with little or no restrictive encoding, and all use the same *control unit*. Another component that they have in common is the *boolean unit*. I will describe these in Section 2.2. Both the control unit and the boolean unit are based on state machines whose specifications are generated by the RL compiler—an approach that is only possible for application-specific processors.

Where Kappa-like architectures vary is in the design of the datapaths. They differ in the functional units and registers provided, and in topology. They do, however, have these features in common: The microoperations that they implement belong to a standard set of microoperation types defined in Section 3.2. The register structure and datapath topology meet basic connectedness requirements to be defined in Section 3.3. And finally, although the above suffices for the RL compiler to be applicable (maybe with crude results), the datapaths of the target family are further constrained to be similar in style to the Kappa datapath. Each is by and large an irregular datapath (one for which good data routing is hard) with a moderate amount of parallelism.

The last proviso is essential if the RL compiler is to be appropriate. There are many ways to design a datapath that is qualitatively different from Kappa's, and that calls for a different kind of compiler. If the functional units are made deeply pipelined, software pipelining becomes the main issue in compilation. If many functional units are added, global optimizations become important. If a regular structure is imposed on the interconnect, a code generation algorithm that fails to take advantage of it does poorly. If the interconnect is pruned so severely that it cannot tolerate minor modifications to the program, any greedy scheduling algorithm risks taking a wrong turn. If the datapath contains a very severe bottleneck, all specialized techniques for generating horizontal code become superfluous, and again a different compiler is called for. These pitfalls are easy to avoid in deriving a datapath from Kappa's, however, and a datapath obtained in this particular way is certainly a Kappa-like datapath, and thus one for which the RL compiler is expected to produce good code.

Section 2.1, which follows, describes the Kappa datapath; some of its derivatives will be described in Chapter 5. Section 2.2 describes the boolean and control units, which are common to all.

2.1 The Kappa datapath

Kappa's datapath is shown in Figure 2.1. Throughout this thesis I will use it as a representative example of the target datapaths.

In the figure, registers are represented by short rectangles, while register banks and memory banks are represented by taller ones. There is one memory bank, on the top left. It differs from the register banks in that reading and writing are mutually exclusive and in that the address can be computed in the datapath.

The registers, the register banks, the memory bank, and one exceptional interconnection near the bottom right of the figure partition the datapath into pipeline stages that are one cycle (one instruction-time) long. In my terminology, there is a one-cycle delay associated with each of the former, and no delay associated with any of the functional units.

The Kappa datapath has two parts. To the right of the `mbus` (the vertical line down the middle of the figure) is the address arithmetic unit. The `mbus` itself and everything to its left constitute the fixed-point arithmetic unit, which will generally have a longer word length.

To avoid wraparound due to overflow, the adder in the fixed-point unit can either do saturating arithmetic or generate a guard bit for use in a subsequent right-shift. In saturating arithmetic, an operation that overflows generates either $2^N - 1$ or -2^N, where N is the word length (not counting the guard bit). The compiler ordinarily uses saturating arithmetic for fixed-point computations.

The intended use of the guard bit is in a multiplication step.[1] Kappa is atypical of signal processors in that it does not include a parallel multiplier. (Some of its derivatives do.) By convention, fixed-point multiplication produces bits $N - 1$ through $2N - 2$ of the signed integer product. Multiplication by a constant is done with an optimal sequence of shifts, additions, and subtractions generated by the compiler. Variable-variable multiplication is done with this sequence of N instructions (which I have written in a more concise C-like syntax than is used by the compiler):

[1] The actual hardware does not handle the guard bit correctly in certain other cases, namely in combination with conditional writing of the `acc`.

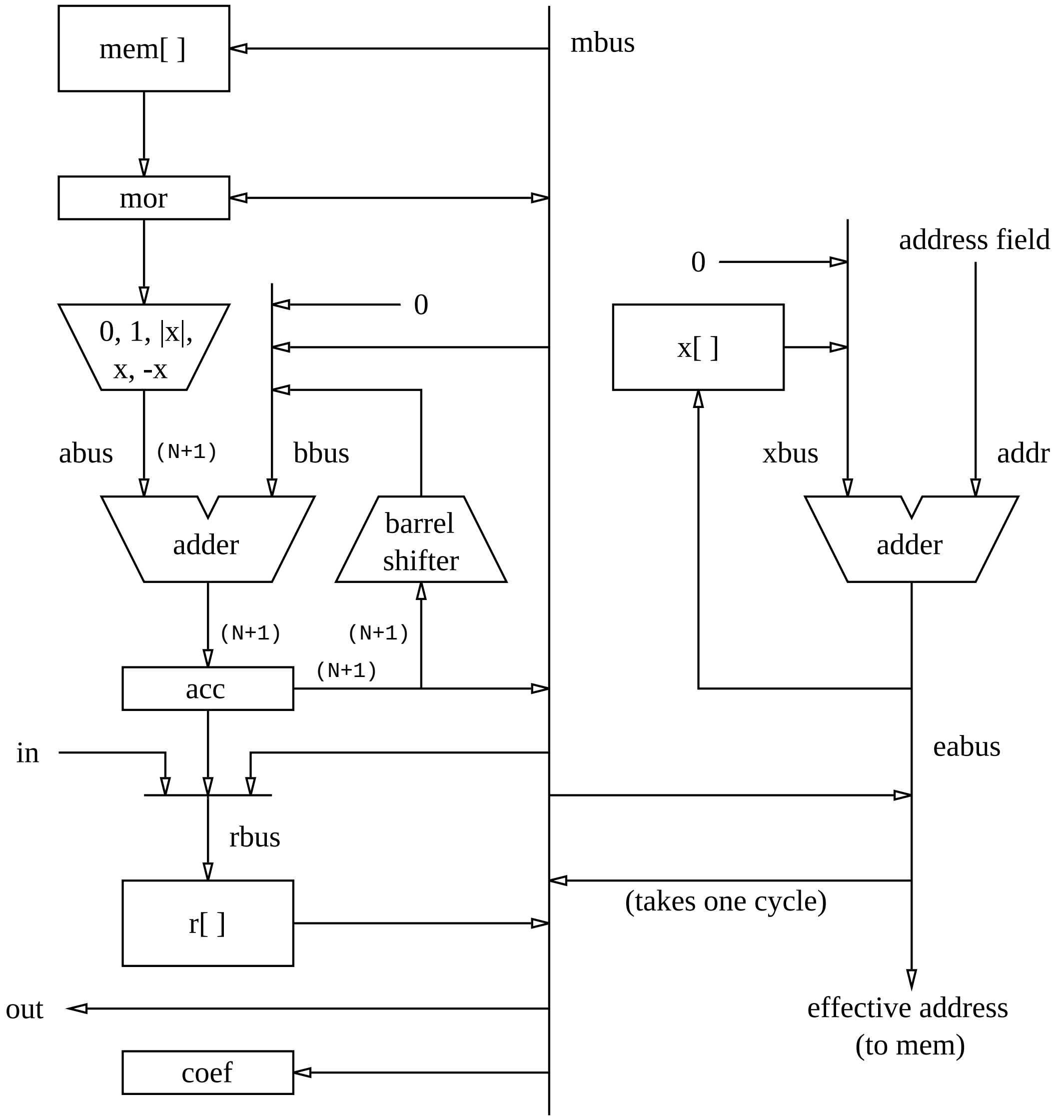

Figure 2.1: The Kappa datapath.

```
     /* Multiplier in coef, multiplicand in mor.  */
1:   acc = (coef & 1) ? mor : 0, coef >>= 1;
2,3,...,N-1:
     acc = (acc >> 1) + ((coef & 1) ? mor : 0), coef >>= 1;
N:   acc = sat((acc >> 1) - ((coef & 1) ? mor : 0));
     /* Product in acc.  */
```

This code uses microoperations that, for simplicity, are not depicted as functional units in Figure 2.1. A more complete description of the Kappa datapath is given in Appendix 1, in the machine description language to be described in Section 3.3.

More important than details about the microoperations that Kappa can execute is the idea that any combination of them can be put into an instruction. For example, Kappa can do input or output (moving the data to memory or from memory via the `mbus`) while it is doing variable-variable multiplication as described above.

2.2 The boolean and control units

The boolean unit and the control unit do the decision-making in Kappa-like application-specific processor architectures. Their designs take advantage of the dedicated nature of the processor; each unit contains a finite state machine whose specification is generated by the compiler.

The boolean unit is like a secondary datapath devoted to the operations `and`, `or`, and `not`. It is used to evaluate the boolean expressions that appear in the program. Because it is actually a state machine based on a programmable logic array, it can simultaneously evaluate any number of complex logical expressions reduced to normal form. The bits of its state represent the values of boolean variables. Its inputs may include signals from off-chip, and sign bits from the adders in the datapath. Its outputs may go off-chip, to the control unit, and to the datapath. In Kappa in particular, one of the boolean unit's outputs (`cc`) can be used to control whether an instruction latches a new value into the accumulator (`acc`).

The control unit generates addresses for the program memory. It contains a state machine and a program counter; the output of the state machine includes a block number, to which the value of the program counter is appended to obtain the address. A branch is performed by simultaneously clocking the state machine and resetting the program counter. This scheme leaves gaps in the program address space, but on an application-specific chip the unused locations can be omitted.

The inputs to the control state machine may include signals from the boolean unit, from off-chip, and from the datapath.[2] With the aid of a return-address stack, the control unit provides a fully general *multi-way jump/call/return* capability based on these primitives:

[2] Signals that go from the datapath to the control unit can always go through the boolean unit at a cost of a one-cycle delay. Eliminating the intermediate step is an optimization that the compiler does not currently perform.

- `goto` *destination-state*
- `call` *destination-state*, *state-to-push*
- `return`

The `return` primitive idles the processor for one cycle so that it can get the destination state from the stack.

Actually, `call` and `return` can be implemented without the return-address stack, if there is no recursion, by using a many-to-one mapping of states to block numbers. The RL compiler does not currently use this approach, even though the RL language prohibits recursion anyway (since there is no stack for local variables).

Chapter 3

The RL Compiler

Ideally the chip designer should not have to know how the RL compiler works. He should be able to treat it as a black box that takes an RL program and a machine description and produces an assembly language program. This chapter describes these three languages.

Originality in language design has not been one of my goals. I have instead striven for simplicity and expedience. Still, the languages may be of interest to those embarking on similar projects.

3.1 RL

RL is loosely based on a subset of C. I tried a subset of Pascal at first, because it has a simple expression syntax and includes a simple counting loop, but later I switched to C as a favor to users who program in C in their day-to-day work.

RL includes significant extensions to C. An RL program cannot be compiled as-is by a standard C compiler, but the RL compiler is capable of translating RL programs into standard C, incorporating calls to a run-time arithmetic library of the user's choice. Indeed, this translation capability, used with a finite-precision library written for the compiler by Lars Svensson, seems to be a useful tool in its own right.

I will assume that the reader is familiar with C and first explain what features are absent from RL. Then I will describe the new features.

Limitations

Many parts of C have been left out of RL for the sake of simplicity:

- There is no separate compilation.
- There are no explicit or implicit function declarations; functions must be defined before they are used.
- Initial values may only be specified in declarations of variables that are to be stored in read-only memory.

- There are no `struct`, `union`, or `enum` types; no `char`, `float`, or `double` types; and no `short`, `long`, or `unsigned`. This leaves only `void`, `int`, pointer types, array types, and the RL-specific types, `bool` and `fix`.
- There are no `goto`, `switch`, `continue`, or `break` statements.
- There is no `typedef`, no `sizeof`, and there are no octal or hexadecimal constants.

Because the target processors do not provide a stack for local variables, the programmer also has to be cognizant of these limitations:

- Recursive function calls are prohibited.
- A register used for a local variable whose scope contains a function call will not be available for use by the compiler elsewhere in the program (on the assumption that the compiler does no analysis of the call graph).

Target architectures need not implement all the arithmetic operations that are in RL. None of those that we have considered so far have implemented integer multiplication, division, or remainder, for example. (But the compiler can always expand multiplication by a literal into other operations.) The policy is that the compiler will not accept programs that use unimplemented operations.[1]

Features borrowed from ANSI C

The recent ANSI standard for C [4] introduces *type modifiers*, namely `const` and `volatile`, and a new preprocessor command, `#pragma`. These have been incorporated into RL. (But my compiler does not do all the error checking that is strictly required for `const` and `volatile`.)

Combinations of type modifiers can be attached to any type. An integer variable, for example, can now be `int`, `const int`, `volatile int`, or `const volatile int`. The meaning of `const` is that the program will not change the value. The meaning of `volatile` is, roughly, as if reads and writes of the value had side effects on the external world, the sequence of which the compiler must preserve.

In RL, the `const` type modifier is used mainly in declaring variables that are to be stored in read-only memory. Currently, the `volatile` type modifier is used mainly to identify boolean variables (variables of type `bool`) that represent external signals. A `volatile bool` variable represents a wire whose level is set by the processor, while a `const volatile bool` variable represents a wire whose level is set externally.[2]

The ANSI standard defines a preprocessor command (or something that looks like one), called `pragma`, that "causes the implementation to behave in an implementation-defined

[1] My compiler is particularly harsh in this regard. If testing for zero is not directly implemented, for example, the programmer who needs to write "`x == 0`" must instead write "`x <= 0 && x >= 0`." This artificial limitation, which I should perhaps remove, is an attempt to bully the programmer into using one or the other inequality.

[2] The program may not change a `const` value, but that is not to say that a `const` value will not change.

manner." In RL, pragmas have the same form as the `define` preprocessor command, but they are allowed to appear only where a top-level declaration is allowed to appear. Pragmas define flags and parameters that control the RL compiler and the Lager system, as in these examples:

word_length determines the number of bits in a processor word (in the case where `int` and `fix` have the same width);

*bank*_**capacity** sets a limit on the number of registers that the compiler can specify for the register bank of the name *bank*.

Features for controlling storage allocation

Register declarations

In C, the programmer can suggest that the compiler store a variable in a register instead of in memory by putting the keyword `register` in front of the declaration. This feature allowed early C compilers to produce good code without using global register allocation. In RL, the `register` keyword serves the same purpose, except that whether the compiler will produce any code at all may be at issue if the datapath is not strongly connected. In RL, unlike in C, the `register` keyword can also be used in top-level declarations and in declarations of `static` local variables.

By default, the RL compiler assigns a variable to a specific memory or register bank according to

- whether or not it is a `register` variable,
- its base type,[3] and
- if the base type is a pointer type, the bank that it points into.

To illustrate, the following table lists the default assignments for the Kappa architecture.

base type	pointer target	normal default	`register` default
`fix`		"`mem`"	"`r`"
`int`		"`mem`"	"`x`"
pointer	"`mem`"	"`mem`"	"`x`"
`bool`		"`bmem`"	"`bmem`"

(Here "`bmem`" refers to the bits of the boolean unit's state.) The defaults for a given architecture are specified by pragmas in the machine description, but can be overridden by pragmas in the RL program. For example, to override the usual defaults for Kappa and store non-`register` integer and pointer variables in "`x`" instead of in "`mem`," the programmer would put the following pragmas into the RL program.

[3]The base type of an array type is the base type of its element type. The base type of a scalar type is itself.

```
#pragma int_memory "x"
#pragma mem_pointer_memory "x"
```

In general, the default bank for a variable of base type *type*, where *type* is `fix`, `int`, or `bool`, is the value of the pragma *type*`_memory` if the variable is not a `register` variable, or of the pragma *type*`_register` if it is a `register` variable. If a variable has base type "pointer to *subtype*," pointing into the bank *bank*, then its default bank is the value either of the pragma *bank*`_pointer_memory` or of the pragma *bank*`_pointer_register`. If *bank* is not specified using register type modifiers as described below, it is assumed to be the same as the default bank for a non-`register` variable of type *subtype*.

Register type modifiers

Assigning variables to default memory and register banks is sometimes too crude. Architectures that have two memory banks, one for each input of a multiplier, are an important case in which this is true. For such cases, RL has *register type modifiers*. A register type modifier is written as the name of a memory or register bank in double quotes. It is a type modifier, like `const` and `volatile`, that can appear wherever `const` and `volatile` can appear. For example, `"foo" int` is a type that describes an integer stored in the bank "`foo`."

I will give some examples because the syntax looks odd. Here is how to declare that the variable `x` has type `"foo" int`:

```
"foo" int x;
```

Pointers present a complication: there are two banks involved, the one in which the pointer resides and the one into which it points. Here is how to declare that the variable `p` points to a `"foo" int`:

```
"foo" int *p;
```

On the other hand, a type modifier for a pointer type is written after the "`*`." Here is how to declare a pointer to `int` that resides in "`bar`" and points into "`foo`":

```
"foo" int * "bar" p;
```

This last flourish would typically not be needed, since the programmer could choose between two default banks for `p` by either adding or not adding the keyword `register` in front of the declaration.

New data types

Booleans

In C, boolean values (*true* and *false*) are represented as integers. In typical general-purpose computers, boolean temporaries are conveniently represented implicitly, through the use of control flow, and boolean variables are indeed conveniently represented as integers. Our application-specific target processors, however, can perform boolean operations upon and store single bits. To determine whether an `int` variable in a C program could be stored in a single bit, a compiler would inconveniently have to consider every use. This is the reason for having a distinct boolean type, `bool`, in RL.

In RL, there are no implicit conversions to or from `bool`, except in certain cases to be described involving literal numbers. *True* can be written as "`(bool) 1`"; *false*, as "`(bool) 0`"; and in most contexts the cast can be omitted. I will now use the term *boolean* to mean "of type `bool`."

The operations that return booleans results are the relationals

`<`, `>`, `<=`, `>=`, `==`, `!=`

and the boolean operations

`&&`, `||`, `!`

The operations that exclusively accept boolean values are the three boolean operations above, and the conditional expression

boolean-condition `?` *else-part* `:` *then-part*

The tests in `if`, `while`, `do-while`, and `for` statements are also required to be boolean.

Fixed-point numbers

RL has a set of fixed-point types. They are a convenience for the programmer who does real arithmetic with integer hardware. He could use `int`, but that approach is clumsy:

- Simple fixed-point constants correspond to huge, impenetrable integer constants.
- The natural multiplication for fixed-point numbers is not integer multiplication. The product of two K-bit numbers[4] can always be represented in $2K - 1$ bits (except for the case of squaring the most negative number). If this must be truncated to J bits ($J = K$ in RL), integer multiplication preserves the lowest J bits while fixed-point multiplication preserves the highest J bits.
- Partly because of this, fixed-point code written using integers tends to be littered with shift operations.

The distinction between integer and fixed-point arithmetic is, moreover, convenient for controlling saturating arithmetic. In RL, fixed-point arithmetic is saturating arithmetic, except in shift operations, and integer arithmetic is always nonsaturating arithmetic.

[4] In 2's-complement representation.

The fixed-point types have names of the form `fix:`n, where n is a possibly negative integer. The form `fix` is a shorthand for `fix:0`. Values of type `fix:`n have a machine-dependent precision and lie in the range

$$2^n > x \geq -2^n$$

Casts may be used to convert between different fixed-point types, but conversions between fixed-point and integer types are not allowed.

The following table shows how the four basic arithmetic operations act on fixed-point numbers.

op	x	y	x *op* y	
+	`fix:`i	`fix:`i	`fix:`i	
-	`fix:`i	`fix:`i	`fix:`i	
*	`fix:`i	`fix:`j	`fix:`$(i+j)$	
/	`fix:`i	`fix:`j	`fix:`$(i-j)$	(unimplemented)

(Since there is no implicit conversion between fixed-point types, if the right-hand side of a `*=` or `/=` has type `fix:`n, then n must be zero.) All of C's floating-point arithmetic operators are available in RL for fixed-point arithmetic; the arguments of a relational operator must have the same type, as must the second and third arguments in a conditional expression.

In addition, the shift operators `<<` and `>>` may be applied to fixed-point values. They perform a signed left-shift or right-shift on the underlying 2's-complement representation of the fixed-point value. The result type is the type of the first argument.

Literal numbers

Every literal number has a well-defined type in C. To obtain this feature in a language with a family of fixed-point types would require inventing new notation for numbers. An entirely different approach is taken in RL: All literal numbers have the type `number`. Thus "`3`" and "`3.0`" are just two different ways of writing the same thing.

Arithmetic on values of type `number` is performed at compile time and produces a result that is also of type `number`. It is an approximation[5] of real arithmetic: unlike in C, the result of evaluating "`1/2`" is one half.

The only implicit conversions that the RL compiler performs are implicit conversions from type `number`. Moreover, it is illegal to implicitly convert a number other than 0 to a pointer type, a number other than 0 or 1 to `bool`, or a non-integer to `int`. Conversions to `bool` occur in these contexts:

- an argument of `!`, `&&`, or `||`,
- the first argument of a conditional (`?:`) expression,
- the test in an `if`, `while`, `do-while`, or `for` statement.

[5]Nothing in the language precludes the use of exact real arithmetic here.

Conversions to `int` occur in these contexts:

- the second argument of `<<` or `>>`,
- the argument of `~`, or either argument of `&`, `|`, `^`, or `%`,
- either argument of `+`, or the second argument of `-`, if the other argument is a pointer,
- the last argument of `in()` or `out()` (described below).

Conversions to types determined by the program occur in these contexts:

- the right-hand side of a simple assignment (not an *op*= assignment),
- an argument of a user-defined function,
- the expression in a `return` statement.

Otherwise, if exactly one argument in a sum, difference, or relational operation has type `number`, it is converted to the type of the other. If exactly one argument in a product or quotient has type `number`, and the other has type `int`, the one of type `number` is converted to `int`. Finally, if exactly one of the last two arguments of a conditional (`?:`) expression has type `number`, it is converted to the type of the other.

Fixed-point multiplication and division are the only operations that can directly combine a value of type `number` with a value of another type. The following table lists those cases.

op	`x`	`y`	`x` *op* `y`	
`*`	`number`	`fix:`i	`fix:`i	
`*`	`fix:`i	`number`	`fix:`i	
`/`	`number`	`fix:`i	`fix:`$(-i)$	(unimplemented)
`/`	`fix:`i	`number`	`fix:`i	

The expressions `2*x` and `x/2`, where `x` is of type `fix`, are common.

In most circumstances, the rules for implicit conversion eliminate the need to put casts on literal numbers, but there are exceptions. For example, the expression "`c ? 1 : 0`" is illegal. (The expression "`c ? (int) 1 : 0`" is legal.)

Miscellaneous features

Predefined functions

RL has three predefined functions: `abs()`, `in()`, and `out()`. The function `abs()` computes absolute value and is overloaded in exactly the same way as unary minus:

```
int abs(x)
  int x;

fix:n abs(x)
  fix:n x;

number abs(x)
  number x;
```

The functions `in()` and `out()` provide input and output for integer and fixed-point values. The programmer is entrusted to correctly specify the type according to which the exchanged data is to be interpreted.

For `out()`, this is the type of the first argument. The second argument, which specifies a machine-dependent port number, is optional and defaults to `0` if omitted.

```
void out(data, port)
  int data;
  int port;     /* a literal integer */

void out(data, port)
  fix:n data;
  int port;     /* a literal integer */
```

For `in()`, the type assumed for the exchanged data is the type to which a literal number would be cast or implicitly converted if it appeared in the same position. (Excepting experimental features, this is the only place in RL where context is used to resolve an overloaded operator or function.) It is illegal for `in()` to appear where it would be illegal for a literal number to appear, or where a literal number would not be cast or implicitly converted to `int` or `fix:`n. The sole argument of `in()`, like the second argument of `out()`, defaults to `0` if omitted.

```
int in(port)
  int port;     /* a literal integer */

fix:n in(port)
  int port;     /* a literal integer */
```

Preprocessor commands

There are four new preprocessor commands: `#repeat`, `#endrepeat`, `#rrepeat`, and `#endrrepeat`. The form

```
#repeat  id N
   text
#endrepeat
```

is roughly equivalent to

```
#define  id 0
   text
#undef  id

#define  id 1
   text
#undef  id

 ...

#define  id N − 1
   text
#undef  id
```

`#rrepeat` and `#endrrepeat` are similar, except that they count backwards. These new preprocessor commands are useful for unrolling and partially unrolling loops.

Program structure

The last difference between RL and C is that the RL programmer can and should leave `main()` undefined. In its place, he defines `loop()` and optionally defines `init()`.

```
void init()
void loop()
```

The compiler then supplies an implicit `main()` of the form shown in Figure 3.1. This is the appropriate form for a signal processing program that is to read a nonterminating input stream.

```
void main() {
  init();   /* omitted if init() is undefined */
  for (;;)
    loop();
}
```

Figure 3.1: The default definition of `main()`.

Figure 3.2 shows a complete RL program, which I will use to illustrate the algorithms of Chapter 4. It is a low-pass filter that smooths an input sequence $x_1, x_2, x_3, \ldots$ to produce an output sequence $y_1, y_2, y_3, \ldots$ using the recurrence

$$y_n = \frac{3}{4}y_{n-1} + \frac{1}{4}x_n$$

There is an example of a more realistic program in Appendix 2.

```
#pragma word_length 16

register fix y;

void init() {
  y = 0;
}

void loop() {
  y = 3/4 * y + 1/4 * (fix) in();
  out(y);
}
```

Figure 3.2: A simple filter described in RL.

3.2 The register-transfer notation

The assembly-language program obtained by compiling Figure 3.2 for Kappa is shown in Figure 3.3. It consists of three sections:

Parameters: definitions of parameters that Lager needs to generate the layout (also, the name of the machine description that the assembler should use).

Data: for each memory and register bank, either a total number of words, or a list of items of the form "*variable*," "*variable*`[`*size*`]`," "*variable* `=` *value*," or "*variable*`[`*size*`]` `=` {*values...*}" (where the last two forms are used for read-only locations).

Code: the microcode, consisting of straight-line segments, each beginning with a label and ending with a control operation (which may be a multi-way jump/call/return).

Each straight-line segment is a sequence of instructions. The register-transfer notation in which these instructions are written is the topic of this section. Of the microoperations that can be written in this notation, only some are implemented by each architecture, but the notation itself is independent of architecture. Indeed, it is also used in the machine description, as described in Section 3.3.

A microinstruction, which I call an instruction, is a set of microoperations and boolean operations (which I will distinguish from other microoperations) separated by commas and terminated by a semicolon. To make the code easier to read, the compiler chains microoperations of the form "... `=` ..." when it can. For example, it abbreviates "`a=b, b=c`," where `b` is a bus, to "`a=b=c`."

Individual microoperations are written as register transfers, in a notation reminiscent of C or RL. Except for variables in address literals (which begin with `&`), identifiers represent

```
/*  "sample.k" compiled for "Vanilla Kappa".  */

.PARAMETERS

arch_file = "vanilla-kappa";
word_length = 12;
stack_depth = 0;
start_state = 0;

.DATA

x       0;
r       2;

.CODE   /* 2 blocks with a total of 9 instructions */

0:      /* 2 instructions */

   /*
    * sample.k, 7: y = (fix ) 0;
    */

    acc=sum=abus=0;
    r[0]=rbus=acc;

                    GOTO 1;

1:      /* 7 instructions */

   /*
    * sample.k, 11: y = (fix ) 0.75 * y + (fix ) 0.25 * (fix ) in(0);
    * sample.k, 12: out(y, 0);
    */

    acc=sum=bbus=mbus=r[0], mor=mbus, r[1]=rbus=in(0);
    abus=mor, acc=sum=bbus+abus, bbus=acc>>1;
    acc=sum=bbus=acc>>1;
    acc=sum=bbus=mbus=r[1], r[1]=rbus=acc;
    mor=mbus=r[1], acc=sum=bbus=acc>>2;
    bbus=acc, abus=mor, acc=sum=sat(abus+bbus);
    r[0]=rbus=acc, mbus=acc, out(mbus,0);

                    GOTO 1;
```

Figure 3.3: The compiled sample program.

nodes in the datapath. These are busses, registers, register banks, memories—places where values can exist.

Subscripting has a special meaning: The name of a register bank or a memory is always followed by a subscript that determines the accessed location. If `r` is a register bank, it always appears in the form `r[I]`, where `I` is an integer. If `mem` is a memory, it may appear in the form `mem[I]`, where `I` can be either an integer or a symbolic address; in the form `mem[x]`, where `x` is a datapath node that holds the effective address; or in the form `mem[I+x]`.

Register transfers specify the times at which values are consumed and produced, using a very general notation together with well-chosen defaults. A node `x` appearing in a register transfer may be annotated with an explicit time offset:

$$\texttt{x.}n$$

Then `x.0` refers to the value of `x` in the current instruction; `x.1`, to the value in the next instruction; `x.-1`, to the value in the previous instruction; and so on. The annotation follows the subscript if one is present:

$$\texttt{x[...].}n$$

A node that appears on the left-hand side of a register transfer of the form "... `=` ...," but not in a subscript, is an output; a node appearing elsewhere is an input. By convention, the time offset n must always satisfy

- $n \geq 0$, for an output, and
- $n \leq 0$, for an input.

The default value of n depends on the position in the register transfer. The default value of n for an input is `0`. The default value for an output is the *delay* of the datapath node—a number specified in the machine description. In other words, a result normally appears in the node into which it is written—becoming available for use as an input in another register transfer—after a number of instruction times equal to the delay of the node. The delay for a bus is, by definition, zero; the delay for an ordinary register is one. (Of the nodes mentioned in the code of Figure 3.3, `acc`, `mor`, and the bank `r` have delays of one, and all the other nodes have delays of zero.)

Besides subscripting (`[]`), timing annotation (`.`), and assignment (`=`), other operator notation in a register transfer is strictly mnemonic. The compiler defines a fixed set of microoperation types with standard meanings. For example, one of these microoperation types is an arithmetic right shift by some number of bits:

```
x = y >> I
```

I will distinguish between *variables*, which stand for datapath nodes, and *parameters*, which stand for integers, by capitalizing the latter. To turn a template such as this into a microoperation, replace the variables with names of datapath nodes, and add timing annotation. A given architecture will implement at most a few of the resulting microoperations—maybe none. Here is one possibility:

```
bbus.0 = acc.0 >> I
```

To obtain a particular instance of a microoperation, replace the parameters with integers (or symbolic address expressions). Here is an instance of the preceding microoperation:

```
bbus.0 = acc.0 >> 2
```

Before describing boolean operations, I will survey the microoperation types supported by the RL compiler. An architecture should implement one or more microoperations for each of these types—with at most a few exceptions. Four types of microoperations are classified as *transfer microoperations*:

```
x = y
x = y[I]
x[I] = y
x = I
```

All other microoperations are *function microoperations.* Among these are indirect read and write,

```
x = y[z]
x[z] = y
```

indexed read and write,

```
x = y[I + z]
x[I + z] = y
```

input and output,

```
x = in(Port)
out(x, Port)
```

nonsaturating (integer) arithmetic,

```
x = -y
x = abs(y)
x = y + z
x = y - z
```

saturating (fixed-point) arithmetic,

```
x = sat(-y)
x = sat(abs(y))
x = sat(y + z)
x = sat(y - z)
```

fixed-point and integer multiplication,

```
x = y * z
x = imul(y, z)
```

bit-wise operations,

```
x = ~y
x = y & z
x = y | z
x = y ^ z
```

and arithmetic shift operations:

```
x = y >> I
x = y << I
x = y >> z
x = y << z
```

There are three overflow-free postshift operations (in which $\mathtt{I} > 0$):

```
x = -y >> I
x = (y + z) >> I
x = (y - z) >> I
```

If k is the word length (the same for `x`, `y`, and `z`), these produce bits `I` through $\mathtt{I} + k - 1$ of the 2's-complement representation of the true integer result of negating, adding, or subtracting, respectively.[6] The compiler uses these three microoperation types in expanding multiplications by constants.

In addition, there are two microoperations types that produce a boolean result,

```
c = (x < 0)
c = (x == 0)
```

and a type that is used to implement conditional expressions (when there are no side effects):

```
x = c ? y : z
```

Finally, there are a few microoperation types used for serial multiplication, but they are likely to change, so I will not list them here.

Pure boolean operations are a special case because they are performed by a programmed logic array designed to perform precisely those operations that appear in the program. They do not conform to any finite set of templates, but are described by this grammar:

[6]In Kappa, because the accumulator has a guard bit, "`bbus.1=abus+bbus>>N`" is equivalent to "`sum=abus+bbus, acc=sum; bbus=acc>>N`."

boolean-operation → *identifier* `:=` `true()`
  | *identifier* `:=` `false()`
  | *identifier* `:=` *exp*

exp → *identifier* | `(` *exp* `)` | `!` *exp* | *exp* `&&` *exp* | *exp* `||` *exp*

The operators `!`, `&&`, and `||` have the same precedences as in C. Identifiers that appear in boolean operations have their own name space, in which only certain datapath node names are reserved—those that appear as `c` in microoperations of these forms:

```
c = (x < 0)

c = (x == 0)

x = c ? y : z
```

An instruction can contain any number of boolean operations.

3.3 The machine description

A machine description consists primarily of definitions of the implemented microoperations, preceded by declarations of the nodes of the datapath. To illustrate, Figure 3.4 shows the portion of Kappa's machine description that describes the address unit. (The Kappa address unit appeared on the right-hand side of Figure 2.1.) The complete machine description is in Appendix 1.

A node declaration begins with the keyword `node` and a list of attribute definitions and flags. The attribute `delay` was described in the previous section; this non-negative integer is the default for the number of cycles after which a value written into the node can be read. The two most important flags are these:

static: A static node is one that retains data from one instruction to the next, until it is overwritten. A bus is an example of a node that is not static.

bank: A bank is a node, such as a register bank or memory, that can hold more than one value at once.

These orthogonal attributes—`delay`, `static`, and `bank`—form a classification system that is more general and more precise than terms like *register*, *latch*, and *bus*.

Like nodes, abstract *resources*, the use of which I will discuss below, must be declared before they are used. A resource declaration simply consists of the keyword `resource`, followed by a list of resource names. These may be chosen arbitrarily, provided that they are distinct from the node names.

A microoperation definition begins with either `micro` or `macro`, defines one microoperation, and may include additional information on indented lines following the first. The `micro` form is the basic one. The `macro` form defines a microoperation as a combination of others.

```
#define bus node:delay=0
#define reg node:delay=1:static
#define file reg:bank

file    x
bus     addr, xbus, xsum, xsign, eabus

micro   addr = Immediate
micro   xbus = x[N]
micro   xsum = addr
micro   xsum = xbus
micro   xsum = addr + xbus
micro   xsum = xbus + addr
micro   xsign = xsum.-1 < 0
micro   eabus = xsum  /* effective address */
micro   x[N] = eabus
```

Figure 3.4: The portion of the Kappa machine description describing the address unit.

A `micro` definition may include information for other programs—the assembler, in particular. Exceptional `micro` definitions may also include compiler directives that impose scheduling constraints. That is, these directives impose additional constraints beyond the fundamental one, which is that two microoperations may not write to the same node at the same time. They come in three varieties:

reserve identifies a node modified by the microoperation but not mentioned in the register transfer—a scratch node. The value written into the node is undefined.

grab identifies an abstract resource to which the microoperation requires exclusive access. The semantics of this is simply that two microoperations may not use the same resource at the same time.

sequence is like `grab`, but creates an additional constraint: the uses of a resource by `sequence` directives must occur in the same order in the compiled code as in the source program.

The `grab` directive should be used when microoperations that do not write a common node are nevertheless incompatible (perhaps because of instruction encoding). The `sequence` directive is used principally to preserve the order of input and output microoperations. (For examples of the use of `grab` and `sequence`, see the references to the resources `read_write` and `input_output` in the Kappa machine description in Appendix 1.)

A `macro` definition includes neither information for the assembler, nor compiler directives. Instead, it specifies a set of previously defined microoperations that together implement the defined microoperation. (The set may be punctuated with both commas and semicolons, so that it resembles a fragment of microcode, if some of the microoperations in the expansion are to be inserted into instructions that follow the current one.) For example, here is the definition of subtraction given in the Kappa machine description:

```
macro sum = bbus - mor
        { abus = -mor, sum = abus + bbus }
```

This defines `sum=bbus-mor` and causes it to invoke the compiler directive "`reserve abus`." In general, a microoperation defined as a macro invokes `reserve` for each intermediate node in the expansion. It also invokes all of the compiler directives invoked by those microoperations (with the proper timing).

Besides `node` and `resource` declarations, and `micro` and `macro` definitions, a machine description can include a few other kinds of entries, about which I have little to say. Some are very simple:

architecture defines the name of the architecture;

pragma defines a default value for a pragma, to be used when the pragma is not present in the RL program.

Others are experimental:

field defines one or more instruction fields (for the assembler);

op associates an RL function name with a user-defined microoperation type.

The machine description should define all the microoperations that the architecture implements of types listed in Section 3.2. Some of these microoperations may be equivalent to combinations of others; these might conveniently be defined as macros. In principle a compiler could aggregate microoperations itself, but the RL compiler does not do this.

There is one exception: composing transfer microoperations. The compiler will automatically compose arbitrarily long sequences of transfer microoperations to chain together function microoperations. In particular, if the machine description defines `a = b` and `b = c`, it need not define `a = c`.

One must pay special attention to transfer microoperations when writing a machine description. First, it is important to define all of them. For example, a machine description that defines the microoperations

```
bbus = 0

sum = abus + bbus

sum = bbus + abus
```

should also explicitly define the microoperation

```
sum = abus
```

Second, unnecessary `reserve` and `grab` compiler directives should not be used when defining transfer microoperations. (The compiler does not allow `sequence` to be used with transfer microoperations at all.) For reasons explained in Chapter 4, violating this rule may reduce the quality of the generated code.[7] A related rule is that transfer microoperations should not be defined as macros. A definition like

```
macro sum = abus    /* Should use micro instead. */
        { bbus = 0, sum = abus + bbus }
```

is bad because it causes `sum=abus` to invoke "`reserve bbus`" although no microoperation that produces a result in `bbus` would ever be scheduled in the same instruction anyway.[8]

Finally, the RL compiler imposes certain requirements on the datapath topology. For a given architecture, the transfer microoperations of the forms

```
x = y
x = y[I]
x[I] = y
```

collectively form a directed graph that may have several disconnected components. The algorithms of Chapter 4 are intended for use with architectures for which each component of the graph contains a node that satisfies these conditions:

- It is static and it is a bank.
- There is a path from it to each node in the component that is read by a function microoperation or by a transfer microoperation of the form `x[I] = y`.
- There is a path to it from each node in the component that is written by a function microoperation or by a transfer microoperation of the form `x = y[I]`, and also from any node to which the program assigns any variables.

This requirement is somewhat stronger than what the compiler strictly needs, but it is probably not restrictive in a practical sense. I will describe the more general requirement, and its origin, in Section 4.3.

[7] Using `reserve` or `grab` in a transfer microoperation creates mutually exclusive groups of place nodes and thus reduces the effectiveness of the spill-path updating algorithm.

[8] I have not looked into having the compiler deduce this, except to observe that it is not trivial. However, the whole issue may be minor—see the discussion at the end of Chapter 5.

Chapter 4

Code Generation Method

The RL compiler divides into a front end and a back end. The front end translates the program into an intermediate representation, partitioning it into straight-line segments. Then, for each straight-line segment, the back end selects microoperations and packs them into instruction words. Generally, only the back end uses the machine description.

The front end performs a variety of routine tasks and simple optimizations, including parsing, constant folding, building the symbol table, and type analysis. In the most interesting optimization, intended to take advantage of multi-way jump/call/return operations, the compiler uses a structured dataflow algorithm to move control-flow operations upward within the program so that they can be coalesced, eliminating all jumps to jumps, jumps to calls, and jumps to returns in the process.

The back end is the focus of this research. This chapter describes the scheduling algorithm that it uses to generate horizontal code for a straight-line segment produced by the front end. I will describe

1. the overall framework, greedy scheduling;
2. the key idea behind my approach, lazy data routing; and finally,
3. the cure for the complications that arise in applying lazy data routing to architectures like Kappa, the spill-path updating algorithm.

4.1 Greedy scheduling

The scheduler's task is to build a fragment of horizontal code from

- a machine description for the datapath and
- a DAG (directed acyclic graph) representing a straight-line segment of the program.

The scheduler must both select microoperations and choose the instructions in which they are to be executed.

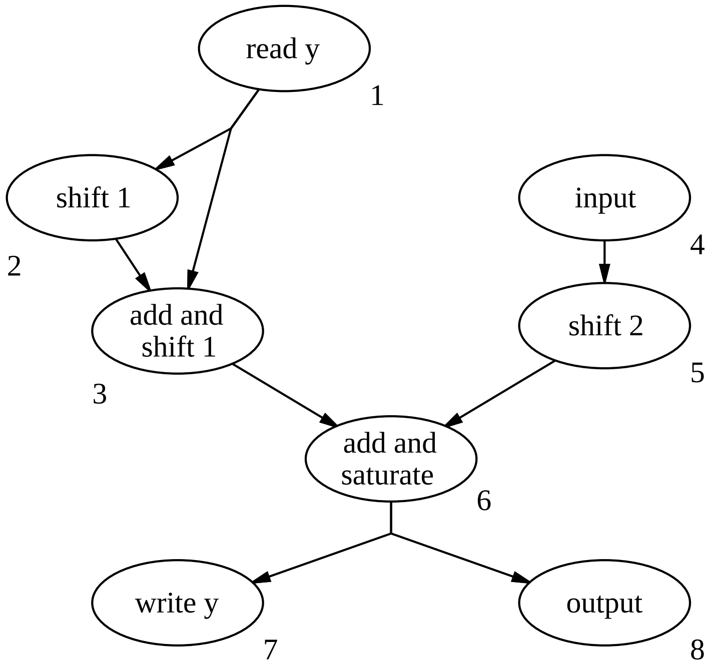

Figure 4.1: The DAG for the loop body in the sample program.

Figure 4.1 shows the DAG for the body of the loop in the sample program of Figure 3.2. The nodes of the DAG represent *operations*, while the edges and forked edges represent *values*. The edges define data dependences constraining the sequence in which operations may be scheduled. Operations that interact with state may additionally be subject to side-effect dependences; for example, there is a read-write dependence here between nodes 1 and 7. This read-write dependence happens to be redundant with the data dependences and is not shown.

Horizontal code does not have to be generated one instruction at a time. Conceptually, the scheduler begins with an infinite sequence of empty instructions, selects microoperations and inserts them into these instructions, and when it is finished, truncates the sequence to its final length. It may well insert microoperations into the second instruction before the first is finalized.

The RL compiler schedules microoperations in groups that correspond to the nodes of the DAG. To schedule a node is to schedule a group consisting of a function microoperation and transfer microoperations to set up its inputs. Once the compiler has successfully scheduled a node, the corresponding microoperations become final; they will not be retracted later in response to difficulties in scheduling other nodes. To impose such a limitation on backtracking is to do *greedy scheduling*.

Before describing the details of scheduling a single node, I want to address the problem of scheduling the DAG. Accordingly, let us presuppose a basic transaction that takes a pair $[N, T]$, where N is a node (whose predecessors have already been scheduled) and $T = 0, 1, 2, \ldots$ is a time, and tries to schedule the node N at time T. The transaction may fail, in which case it has no effect.

Scheduling the DAG

For each node N of the DAG, the scheduler must find some time T at which N can be scheduled. It does this by trying a succession of pairs $[N, T]$, never retracting a successful transaction. Thus the problem comes down to choosing what pair $[N, T]$ to try next. The choice is from among untried pairs $[N, T]$ such that

- N has not been scheduled,
- all nodes on which it depends have been scheduled, and
- none of these was scheduled at a time later than T.

If N directly depends on N', which has been scheduled at time T', scheduling N at time T is not ruled out for $T = T'$; only for $T < T'$. This is useful for modeling architectures like Kappa. If this is a data dependence, it may still cause the transaction $[N, T]$ to fail, depending on the time at which N' writes its output, the time at which N reads its input, and the time needed to route the data from one place to the other. If this is a side-effect dependence between read or write nodes that potentially access the same location, the constraint imposed on their relative positions in time is, by design, exactly the right one; regardless of the definitions of the corresponding transfer microoperations, internally the compiler uses these conventions for read and write nodes:

- A read node scheduled at time T fetches the value that the variable has at time T.
- A write node scheduled at time T sets the value that the variable has at time T.

The choice of the pair $[N, T]$ to try next is guided, in part, by priority orderings for the nodes N and the times T. For times, the natural ordering gives priority to earlier T. For nodes, there are several popular ideas. One can give higher priority to

- nodes (i.e., operations) that appear earlier in the source program,
- nodes that appear higher in the DAG (where height means execution time of the longest path to a leaf), or
- nodes of particular types, such as those that must not be scheduled in the last instruction of the sequence.

The first strategy tends to keep the lifetimes of temporaries short. The second, recommended by Fisher [15], is effective when one expects the DAG's critical path to determine the number of instructions. The third is analogous to good RISC-instruction reordering strategy: giving high priority to instructions with delay slots to increase the opportunities for filling them. I think the most effective choice for the RL compiler would be a blend of these three strategies. For simplicity, however, the RL compiler uses source-order priority. The priority ordering for the sample DAG is the order in which the nodes in the figure are numbered.

Now assume N_1 has higher priority than N_2, and $T_1 < T_2$. In choosing the next scheduling transaction to try, we would choose $[N_1, T]$ over $[N_2, T]$ for any T, and we would choose $[N, T_1]$ over $[N, T_2]$ for any N. But between $[N_1, T_2]$ and $[N_2, T_1]$ the choice is less clear. We could put more weight on the node, or on the time. There are two basic variants of greedy scheduling:

- *List scheduling* chooses $[N_2, T_1]$, putting more weight on the time.
- *Operation scheduling* chooses $[N_1, T_2]$, putting more weight on the node.

List scheduling and operation scheduling are almost equivalent. In fact, they produce identical schedules when applied to the problem of scheduling tasks with precedence constraints, resource constraints, and preassigned scheduling priorities such that

- each task completes in one unit of time,
- there are no resource constraints between tasks that execute at different times, and
- the scheduling priorities assigned to the tasks are consistent with the precedence constraints (which are analogous to the DAG).

List scheduling, however, may be easier to program for that problem. This is because list scheduling completely determines the set of tasks to be executed in one time interval before beginning to build the set to be executed in the next.

For the problem at hand, however, list scheduling has no significant advantage over operation scheduling. It would be nice if the RL compiler could generate one instruction at a time, but the nature of lazy data routing is such that scheduling a node at time T may involve scheduling transfer microoperations in much earlier instructions.

I have chosen to use operation scheduling. Because I use a priority ordering that is a total ordering of the DAG, the algorithm takes a particularly simple form that does not require the compiler to actually construct the DAG. This algorithm is shown in Figure 4.2.

Scheduling a DAG node

In trying to schedule a given node of the DAG at a given time, the compiler may have several different function microoperations of the same type to consider. For example, the Kappa machine description defines four ways to add a pair of integers:

```
sum = abus + bbus
sum = bbus + abus
xsum = addr + xbus
xsum = xbus + addr
```

for each node N, in order of priority **do**
  $T_0 \leftarrow max\{time(N') \mid N \text{ depends on } N'\}$;
  for $T \leftarrow T_0, T_0+1, T_0+2, \ldots$ **do**
    try to schedule N at T;
    if successful **then**
      break;

Figure 4.2: Scheduling the DAG.

The first and second arguments could be taken from `abus` and `bbus` respectively, or vice versa, or from `addr` and `xbus` respectively, or vice versa. To schedule a node N at a time T, the compiler just tries all applicable function microoperations in the order in which they appear in the machine description.

This reduces the problem to that of scheduling a given function microoperation at a given time T. Assume that the microoperation has the form

$$x_0.t_0 \texttt{ = f(} x_1.t_1\texttt{, } x_2.t_2\texttt{, ..., } x_n.t_n\texttt{)}$$

where x_i is the name of a datapath node and t_i is an integer for $i = 0, 1, 2, \ldots, n$. This microoperation takes n argument values $v_1, v_2, \ldots, v_n$ and produces one result value v_0. Scheduling it at time T involves these steps:

1. For each v_i, *deliver* v_i to x_i at time $T + t_i$; that is, find a sequence of transfer microoperations that propagates v_i to x_i at time $T + t_i$ from some place and time where it already exists. (Section 4.2 will describe the data routing algorithm that does this.)

2. Put the function microoperation in instruction T and verify the constraints due to `reserve`, `grab`, and `sequence` compiler directives associated with it in the machine description.

3. Record the presence of the new value v_0 in x_0 at time $T + t_0$.

If any step is inconsistent with the current schedule, some other microoperation or time must be tried.

Actually, a little more persistence is called for. An unfortunate choice among ways of delivering one argument may preclude the delivery of another. If delivery of an argument other than the first fails, it seems to be a good idea to try again, delivering that argument first. So that time is not wasted on lost causes, however, the procedure should incorporate this preliminary step:

0. For each v_i, check that the cost estimate for delivering v_i to x_i at time $T + t_i$ is finite. (See Section 4.2.) Check that the resources required by compiler directives are available. Check that x_0 is empty at time $T + t_0$.

I will summarize the entire procedure more formally in Section 4.2, in Figures 4.10 and 4.9.

That completes the explanation of how to schedule an ordinary node of the DAG. "Read" and "write" nodes and nodes that generate constants, however, are special and are scheduled differently. I will describe the details, but they are not essential for reading the rest of the chapter.

Consider, first, reading or writing a location that is known at compile time. Accessing a scalar variable is an example of this, as is accessing an array using a constant as the index. To schedule a "write" node of this sort, the compiler applies the usual procedure to the corresponding microoperation (as if it were a function microoperation), except that in step 3 it puts the input value, rather than a new value, into the destination. To schedule a "read" node, the compiler does nothing more than return the last value that a "write" node put into the given location, or the initial value that the compiler put there when the scheduling of the DAG was begun. The corresponding microoperation will be used in delivery routes for nodes that use this value.

Generating a constant is like reading from a fictitious read-only register. By scheduling nodes that generate constants like it schedules "read" nodes, the compiler reaps substantial benefits. When there are two uses of a constant, a choice presents itself: The constant can be generated once and used twice, or it can be generated twice. In the process of choosing among delivery paths, the RL compiler's data routing algorithm makes both this choice and the choice among different ways to generate the same constant.

Finally, consider reading or writing a memory location whose address is not known at compile time. Such a node is scheduled with the usual procedure, skipping step 1 if reading, or skipping step 3 if writing. After scheduling a "write" node, the compiler must update its record of the latest value in each location in the given memory bank. For each location that could potentially be the one being written, the recorded value must be replaced to reflect the possibility that the actual value has changed. Determining the set of locations to which a "read" or "write" node might refer is *alias analysis*, which is very difficult in general. All that the RL compiler tries to do is find nodes for which a particular location or array can be identified as the target.[1] It uses this information also to determine read-write, write-read, and write-write dependences for use in the algorithm of Figure 4.2.

Example

Now I will show how the compiler generates code for the sample DAG, which I repeat here in Figure 4.3. It will be necessary to refer to the diagram of Kappa in Figure 2.1.

The first step is to create a value token for the initial value of the variable `y`, which has been assigned to register 0 in the register bank `r`. This will be value 1. I will consecutively number other values as they are created. Because the nodes of the DAG that will create new values are nodes 2–6, it happens that subsequent values will have the same numbers

[1] If `p` is a pointer into an array and `i` is an `int`, the expression `*(p+i)` is illegal unless it refers to an element of the same array. The RL compiler can thus conclude, for example, that the expression `a[i]` (where `a` is an array) refers to an element of `a`.

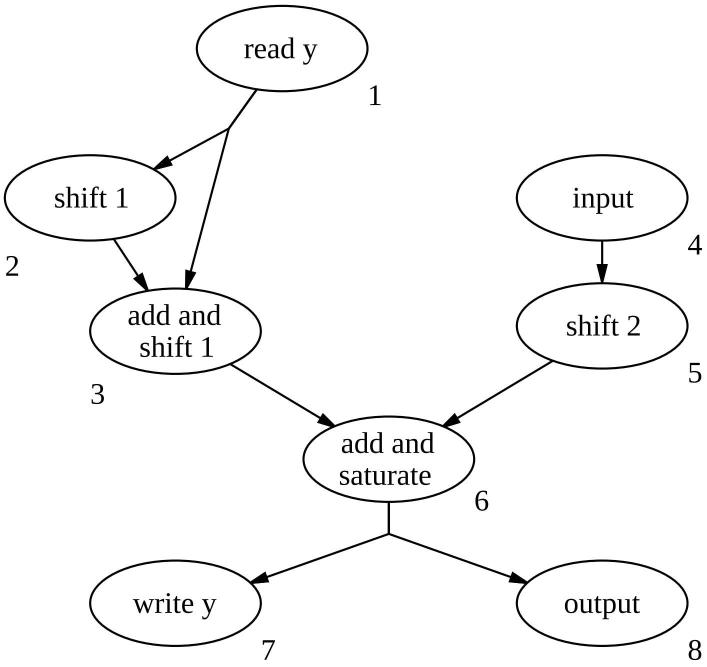

Figure 4.3: The sample DAG (also shown in Figure 4.1).

as the nodes that create them.

Scheduling node 1 of the DAG, "read `y`," is easy; the compiler just looks up the current value of `y`, value 1, without scheduling any microoperations. Value 1 becomes both the input to node 2 and one of the inputs to node 3.

Node 2 is the operation "shift 1," for which there is the function microoperation `bbus=acc>>N`. The attempt to schedule this node in the first instruction fails for lack of a delivery route, but the compiler succeeds in scheduling it in the second instruction. The result, value 2, is available on the `bbus` in the same instruction. After node 2 is scheduled, only the first two instructions are non-empty:

```
acc=sum=bbus=mbus=r[0];
bbus=acc>>1;
```

Node 3 is the operation "add and shift 1," for which there are the microoperations `bbus.1=(abus+bbus)>>N` and `bbus.1=(bbus+abus)>>N`, which are defined as macros. The inputs to node 3 are value 2 and value 1, in that order. The compiler tries to schedule `bbus.1=(abus+bbus)>>N` in the second instruction, but is unable to deliver value 2 to the `abus`. It succeeds, however, in scheduling the other version, `bbus.1=(bbus+abus)>>N`, in the same instruction. The delivery route for value 2 has zero length. The delivery route for value 1 begins in the middle of the previous route. After node 3 is scheduled, the code looks like this:

```
acc=sum=bbus=mbus=r[0], mor=mbus;
bbus=acc>>1, abus=mor, acc=sum=bbus+abus;
bbus=acc>>1;
```

The result of node 3, value 3, is available on the `bbus` in the third instruction. Since the `bbus` is indeed a bus, value 3 will disappear unless something is done with it. The idea of lazy data routing is to decide what to do with it later, when the node that uses it, node 6, is scheduled. A reason to worry is that the compiler might schedule nodes 4 and 5 in such a way that all delivery routes for value 3 are blocked, so that node 6 cannot be scheduled in any instruction. Making sure that this does not happen is the hard part of the compiler's job: at each step in the scheduling of nodes 4 and 5, it must check that value 3 has a *spill path*, a hypothetical delivery route to the indefinite future. This mechanism will be described in Section 4.3, but its function will be apparent here.

Node 4 is the operation "input"—actually "input 0"—for which there is the microoperation `rbus=in(Port)`. It can be scheduled in the first instruction; there is a spill path for the result (value 4) that does not conflict with the spill path for value 3.

```
acc=sum=bbus=mbus=r[0], mor=mbus, rbus=in(0);
bbus=acc>>1, abus=mor, acc=sum=bbus+abus;
bbus=acc>>1;
```

For node 5, "shift 2," there is again the function microoperation `bbus=acc>>N`. Scheduling this node is interesting. The compiler's attempts to schedule it in the first, second, and third instructions fail for obvious reasons. The attempt to schedule it in the fourth instruction fails because there is no delivery route for the argument (value 4) that leaves open a spill path for value 3. The compiler finally succeeds in scheduling node 5 in the fifth instruction, choosing one of several plausible delivery routes:

```
acc=sum=bbus=mbus=r[0], mor=mbus, r[T1]=rbus=in(0);
bbus=acc>>1, abus=mor, acc=sum=bbus+abus;
bbus=acc>>1;
acc=sum=bbus=mbus=r[T1];
bbus=acc>>2;
```

For node 6, "add and saturate," there are the microoperations `sum=sat(abus+bbus)` and `sum=sat(bbus+abus)`. Because of difficulties in delivering value 3, the earliest that that this node can be scheduled is the sixth instruction[2]:

```
acc=sum=bbus=mbus=r[0], mor=mbus, r[T1]=rbus=in(0);
bbus=acc>>1, abus=mor, acc=sum=bbus+abus;
acc=sum=bbus=acc>>1;
acc=sum=bbus=mbus=r[T1], r[T2]=rbus=acc;
mor=mbus=r[T2], acc=sum=bbus=acc>>2;
bbus=acc, abus=mor, sum=sat(abus+bbus);
```

[2]Node 6 could be scheduled one instruction earlier if the compiler had chosen a different delivery route for the argument of node 5.

After the compiler has scheduled the remaining two nodes of the DAG, it does local register allocation for the part of each register bank or memory bank used for temporary storage. Here is the end result:

```
acc=sum=bbus=mbus=r[0], mor=mbus, r[1]=rbus=in(0);
bbus=acc>>1, abus=mor, acc=sum=bbus+abus;
acc=sum=bbus=acc>>1;
acc=sum=bbus=mbus=r[1], r[1]=rbus=acc;
mor=mbus=r[1], acc=sum=bbus=acc>>2;
bbus=acc, abus=mor, acc=sum=sat(abus+bbus);
r[0]=rbus=acc, mbus=acc, out(mbus,0);
```

4.2 Lazy data routing

Data routing is the task of finding delivery routes. For describing it, and for carrying it out, the *place graph* and the *place-time graph* are helpful. I will describe these data structures, and then the data routing algorithm.

The place graph

The place graph is a directed graph that summarizes the transfer microoperations that can be used in delivery routes. The edges of the graph are labeled with the amounts of time used by the corresponding transfer microoperations. The nodes of the graph, which I will call *places*, are a superset of the nodes of the datapath. Places have two attributes: *capacity* and *width*. The capacity is the number of values that the place can hold. The width is the maximum number of bits in a value.

Figure 4.4 shows the place graph that the compiler uses in compiling the sample DAG. The plain circles represent places that can hold only one value; the double circles represent places with unlimited capacity; and it happens that no places with intermediate capacity appear. Because zero is such a common edge label, I have included only non-zero edge labels in the figure.

The boxes in the figure identify groups of mutually exclusive places nodes. Only unit-capacity nodes may belong to such a group, which in effect has unit capacity itself: only one node at a time may hold a value.[3]

Groups of mutually exclusive place nodes arise from transfer microoperations that have `reserve` and `grab` compiler directives associated with them. The compiler breaks such transfer microoperations into multiple steps, creating an intermediate place node for each compiler directive. Intermediate nodes that arise from `grab` directives for a common resource are made mutually exclusive. Intermediate nodes that arise from `reserve` directives for a common node are made mutually exclusive with it and with each other. The lower mutually exclusive pair in the figure arises from these microoperation definitions:

[3]Because the compiler enforces mutual exclusion internally by substituting a collective capacity limit for the individual limits, groups of mutually exclusive place nodes must always be disjoint.

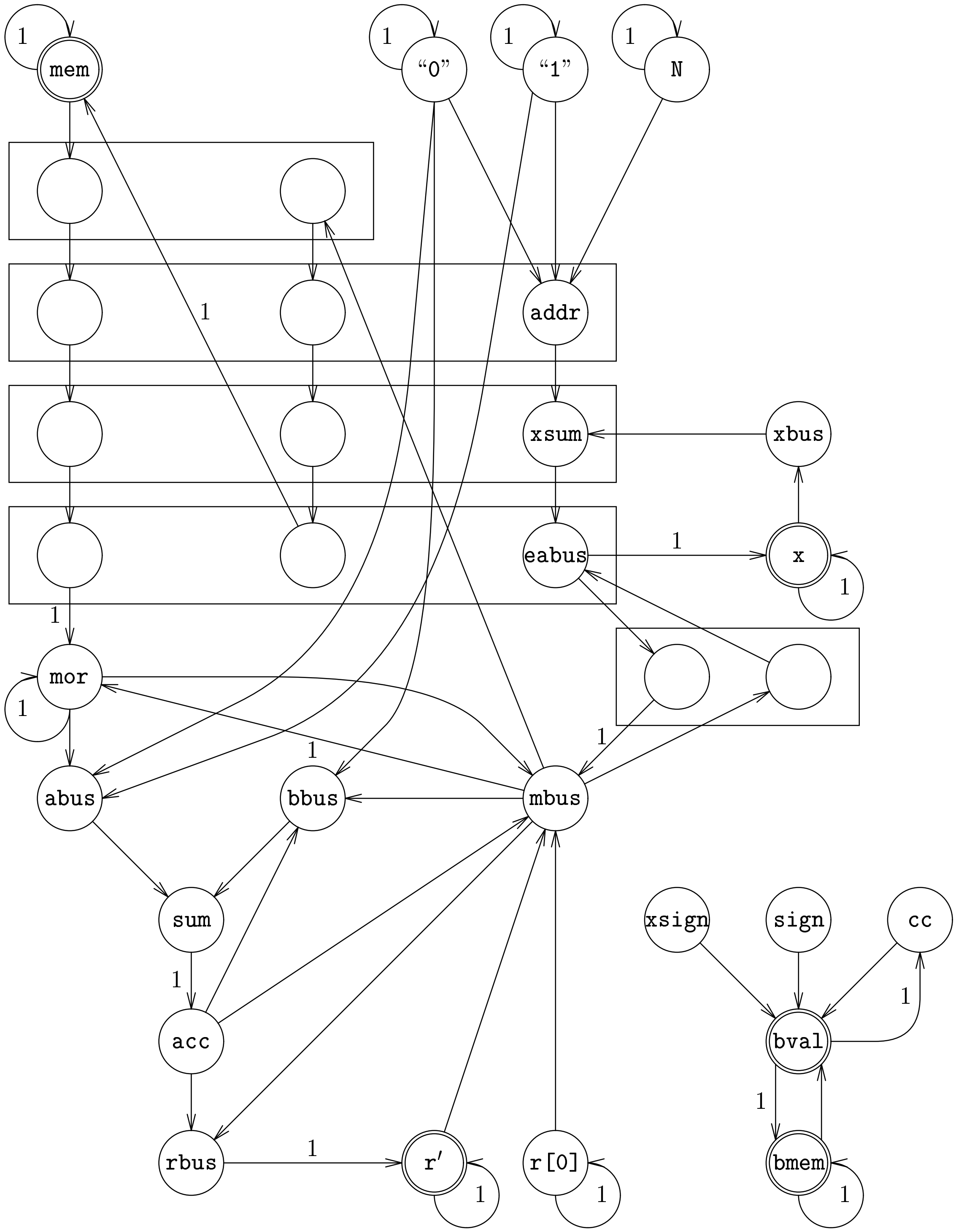

Figure 4.4: The place graph for Kappa, with a variable assigned to `r[0]`. The disconnected part on the lower right represents the boolean unit.

```
micro eabus = mbus
        grab one_way_only

micro mbus.1 = eabus
        grab one_way_only
```

The other four groups of mutually exclusive place nodes arise from these microoperation definitions:

```
macro mor = mem[Loc]
        { addr=Loc, xsum=addr, eabus=xsum, mor=mem[eabus] }

macro mem[Loc] = mbus
        { addr=Loc, xsum=addr, eabus=xsum, mem[eabus]=mbus }
```

Both `mor=mem[Loc]` and `mem[Loc]=mbus` evidently implicitly reserve `addr`, `xsum`, and `eabus`.[4] They also grab the resource `read_write`, since they are defined in terms of the microoperations `mor=mem[eabus]` and `mem[eabus]=mbus`, which are themselves defined as follows:

```
micro mor = mem[eabus]
        grab read_write

micro mem[eabus] = mbus
        grab read_write
```

In scheduling a function microoperation as described in the previous section, a `reserve` or a `grab` just directs the compiler to store a dummy value in a particular place node (i.e., to label a place-time node as described below). For a `reserve`, this is naturally the reserved node. For a `grab`, this is an arbitrarily chosen member of the group of mutually exclusive intermediate nodes that was created for the grabbed resource. If the resource is grabbed only by function microoperations, and hence no intermediate node was created for it, a dummy place node is created. There are a few such dummy nodes in the place graph of the figure, but they are not shown.

The other nodes in the figure that do not correspond to datapath nodes defined in the machine description are "`0`", "`1`", and `N` (at the upper right), and `r[0]` (at the bottom). The first three are fictitious registers that hold constants, as described in the previous section; the node "`0`" holds 0, the node "`1`" holds 1, and the node `N` holds other pointer and integer constants. The fourth place node, `r[0]`, represents the location to which the variable `y` has been assigned. Unlike the place node `r'`, which represents all the other locations in the bank, `r[0]` has no incoming edge. No delivery route should enter it.

[4] In Section 3.3 I advised against defining transfer microoperations as macros, but I have defined these two as macros anyway because it makes the definitions much clearer.

The place-time graph

The place-time graph is a directed graph whose nodes are of the form $[P, T]$ where P is a place and $T \in \{0, 1, 2, \ldots\}$ is a time. The place-time nodes $[P_1, T_1]$ and $[P_2, T_2]$ are connected by an edge if P_1 and P_2 are connected in the place graph by an edge labeled $T_2 - T_1$. To illustrate, Figures 4.5 and 4.6 show a simple place graph and the corresponding place-time graph.

A place-time node may be labeled by one or more values: $[P, T]$ is labeled by the value v if v is available in P at time T. Labels are attached to nodes by two different mechanisms. First, the scheduling of function microoperations introduces newly computed values into the graph. Second, data routing copies values from node to node along its edges.

Place-time nodes inherit the attributes of place nodes:

$$\begin{aligned} \mathit{capacity}([P, T]) &= \mathit{capacity}(P) \\ \mathit{width}([P, T]) &= \mathit{width}(P) \end{aligned}$$

Similarly, $[P_1, T], [P_2, T], \ldots, [P_n, T]$ are mutually exclusive if $P_1, P_2, \ldots, P_n$ are mutually exclusive.

Various circumstances can preclude labeling a place-time node. First, there are basic conditions that a node must satisfy if it is to be labeled with any additional value whatsoever. I will call a node *open* if it satisfies these. A node $[P, T]$ is open if

- the number of labels that it has is less than its capacity, and
- it is not mutually exclusive with a labeled node.

A place-time node cannot accept an additional label unless it is open, but that is not the only condition. Specifically, a place-time node Q can be labeled with an additional value v if it is open, if $\mathit{width}(Q) \geq \mathit{width}(v)$, and if there will still be spill paths afterwards for all values that need them. This is summarized as a procedure in Figure 4.7.

The labels of a place-time node $[P, T]$ constitute a set, denoted $\mathit{labels}([P, T])$. They do not have a particular sequence; the compiler accumulates values to be stored in P at time T, but does not assign them to particular locations (if $\mathit{capacity}(P) > 1$). It must ultimately do that, however, respecting the rule that a value

$$v \in \mathit{labels}([P, T]) \cap \mathit{labels}([P, T + 1])$$

must be assigned to the same location at time $T + 1$ as at time T. Once the final set $\mathit{labels}([P, T])$ is known for all T, it is easy make the assignments in order of increasing T.

To see that it is impossible to optimally assign values to registers on the fly, consider a time series of three place-time nodes of capacity 2. Label the first with value 1. Label the third with value 2, deciding whether to use the same location or the other location. If the other, then try to label all three nodes with value 3. If the same, label the first and second with value 3, and then try to label the second and third with value 4.

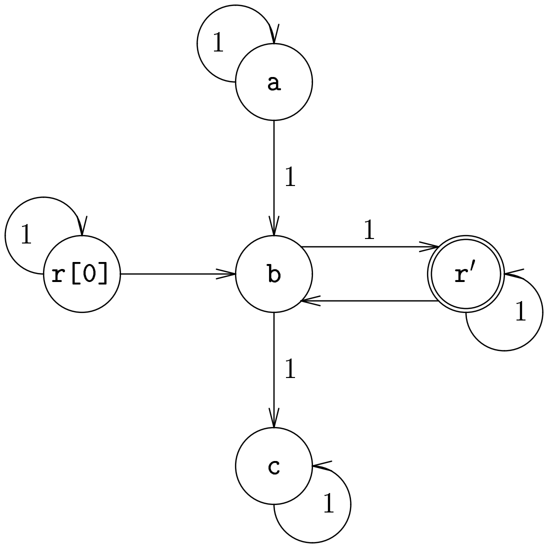

Figure 4.5: A simple place graph, for a datapath with a bus b, registers a and c, and a register bank r. The node for r has been split because a variable has been assigned to r[0]. (This datapath will also serve as an example in Section 4.3.)

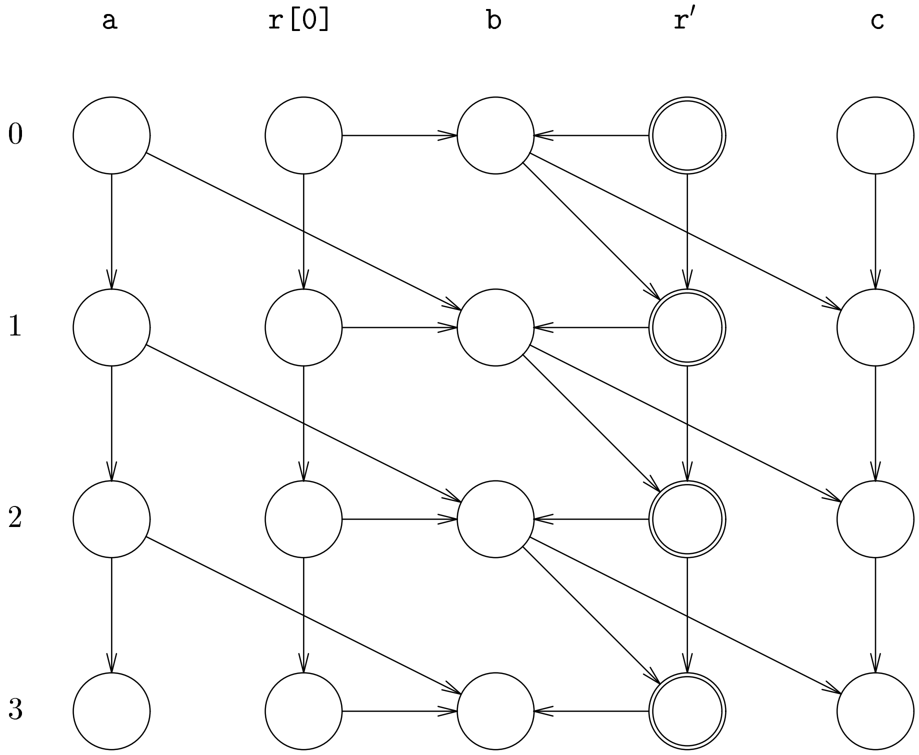

Figure 4.6: The corresponding place-time graph, truncated at $T = 3$.

function *attach-label*(v, Q) **is**
  if $v \in \mathit{labels}(Q)$ **then**
    return *success*;
  if $\mathit{open?}(Q) = \mathit{false}$ or $\mathit{width}(Q) < \mathit{width}(v)$ **then**
    return *failure*;
  return *actually-attach-label*(v, Q);

function *open?*(Q) **is**
  return *effective-capacity*$(Q) > 0$;

function *effective-capacity*(Q) **is**
  if $\exists Q'$, $\mathit{labels}(Q') \neq \emptyset$ and Q' is mutually exclusive with Q **then**
    return 0;
  return $\mathit{capacity}(Q) - |\mathit{labels}(Q)|$;

function *actually-attach-label*(v, Q) **is**
  $\mathit{labels}(Q) \leftarrow \mathit{labels}(Q) \cup \{v\}$;
  if there is a consistent set of spill paths for the live values **then**
    return *success*;
  $\mathit{labels}(Q) \leftarrow \mathit{labels}(Q) - \{v\}$;
  return *failure*;

Figure 4.7: The procedure for trying to label a node. Section 4.3 will revise the definition of *actually-attach-label*.

```
function deliver(v, Q) is
    current-mark ← a new mark;
    return deliver-aux(v, Q);

function deliver-aux(v, Q_0) is
    mark(Q_0) ← current-mark;
    if v ∈ labels(Q_0) then
      return success;
    if attach-label(v, Q_0) = success then
      S_1 ← { Q ∈ predecessors(Q_0) | mark(Q) ≠ current-mark };
      S_2 ← { [Q, C] | Q ∈ S_1 and C < ∞, where C = cost-estimate(v, Q) };
      for [Q, C] ∈ S_2, in order of increasing C do
          if deliver-aux(v, Q) = success then
            return success;
      undo the effects of the "attach-label(v, Q_0)";
    return failure;
```

Figure 4.8: The data routing algorithm. Section 4.3 will revise the definition of *deliver*.

Finding delivery routes

A *potential delivery route* for a value v and a destination $[P, T]$ is a path consisting of distinct place-time nodes $Q_1, Q_2, \ldots, Q_n$ $[n \geq 1]$ such that

- Q_1 is labeled by v,
- $Q_2, Q_3, \ldots, Q_n$ are open, and
- $Q_n = [P, T]$.

The function of the data routing algorithm is to try to find a potential delivery route whose nodes can actually be labeled with v.

Mutual exclusion and spill paths have this effect: Whether a place-time node can be labeled with a value v may depend on the labels of other nodes. This means that a search that visits each node at most once may fail to find an existing delivery route. Nevertheless, the way in which the compiler looks for a route is with a depth-first search rooted at $[P, T]$. Nodes are labeled with v as the search goes forwards (following edges of the place-time graph *backwards*) and unlabeled as it backs up. The search terminates as soon as a node already labeled by v is found. The penalty for using this heuristic algorithm is an occasional, unnecessary failure to find a route. Such a failure will at worst cause the compiler to schedule an operation later than necessary.

The data routing algorithm is shown in Figure 4.8. It uses *delivery cost estimates* for the predecessors of a node to choose the order in which to traverse the node's incoming edges. The estimate used for the cost of delivering the value v to the node Q is the cost of the cheapest potential delivery route that is wide enough for v, where the cost of the potential route $Q_1, Q_2, \ldots, Q_n$ ($Q_n = Q$) is defined to be

$$\sum_{i=2,3,\ldots,n} cost(Q_i)$$

The function *cost* is defined in terms of a set of given place costs:

$$cost([P,T]) = cost(P)$$

The compiler calculates $cost(P)$ using a somewhat arbitrary heuristic, basing its value on the capacity and delay of P, and on whether P is in a mutually exclusive group.

Delivery cost estimates are described by this data flow equation:

$$cost\text{-}estimate(v,Q) = \begin{cases} 0 & \text{if } v \in labels(Q) \\ \infty & \text{if } v \notin labels(Q) \text{ and } Q \text{ is not open or } width(Q) < width(v) \\ \multicolumn{2}{l}{cost(Q) + min\{cost\text{-}estimate(v,Q') \mid Q' \in predecessors(Q)\}} \\ & \text{otherwise} \end{cases}$$

Before executing $deliver(v, [P,T])$, the compiler will have precomputed

$$cost\text{-}estimate(v, [P', T'])$$

for all P' and all $T' <= T$, using a straightforward data-driven relaxation algorithm that has been optimized for the common case in which the place-time graph is acyclic. In fact, the compiler computes these estimates far ahead of time. It precomputes the cost estimates for all the input values of a node of the DAG before an attempt to schedule the node (building on any estimates computed in previous attempts to schedule the same node). The estimates used by the data routing algorithm may consequently be slightly out-of-date, but an estimate will never falsely indicate that a delivery is impossible.

Delivering more than one value

For delivering n argument values—v_1 to Q_1, v_2 to Q_2, $\ldots$, v_n to Q_n—a reasonable procedure is to call $deliver(v_i, Q_i)$ for $i = 0, 1, \ldots, n$, in that sequence. However, as stated in the previous section, trying other sequences can be helpful in occasional, troublesome cases. Figure 4.9 shows the procedure used in the RL compiler. It looks expensive, but in practice n is rarely as large as three. It is used in the procedure for scheduling a function microoperation, which is shown in Figure 4.10.

4.3 Spill paths

The previous section's definition of a *potential delivery route* will be the starting point here. Given a distinguished set of place-time nodes, termed *safe*, a *spill path* for a value v is a potential delivery route $Q_1, Q_2, \ldots, Q_n$ to a *safe* place-time node Q_n, such that

```
function deliver-multiple([v1, v2, ..., vn], [Q1, Q2, ..., Qn]) is
    for i ← 1, 2, ..., n do
        if deliver-and-kill(vi, Qi) = failure then
            return failure;
        if deliver-multiple([v1, ..., v(i-1), v(i+1), ..., vn],
                            [Q1, ..., Q(i-1), Q(i+1), ..., Qn]) = success then
            return success;
        undo the effects of the "deliver-and-kill(vi, Qi)";
    return failure;

function deliver-and-kill(v, Q) is
    if deliver(v, Q) = failure then
        return failure;
    if v will never be delivered elsewhere (except after backing up) then
        make-dead(v);
    return success;
```

Figure 4.9: Delivering a set of argument values.

- no two nodes Q_i and Q_j are mutually exclusive, and
- $\mathit{width}(Q_i) \geq \mathit{width}(Q_{i-1})$ for $i = 2, 3, \ldots, n$.

The condition $\mathit{width}(Q_i) \geq \mathit{width}(Q_{i-1})$ is used instead of $\mathit{width}(Q_i) \geq \mathit{width}(v)$ because the algorithms that I will describe require that a partial spill path can be extended without knowing the value.

When several values must have spill paths, it would not suffice to consider them individually. Define a set of spill paths—one for each value in a set of values—to be *consistent* if it satisfies these conditions:

- No node on one path is mutually exclusive with a node on another.
- The number of spill paths on which a node appears does not exceed the node's *effective capacity*:

$$\mathit{effective\text{-}capacity}(Q) = \begin{cases} 0 & \text{if } Q \text{ is not open} \\ \mathit{capacity}(Q) - |\mathit{labels}(Q)| & \text{otherwise} \end{cases}$$

Define a *live* value to be a value that labels at least one place-time node and still has a delivery pending or in progress. The compiler is based on this rule: There must always be a consistent set of spill paths for the live values.

```
v0 ← a new value to represent the result;
for i ← 0, 1, ..., n do
    Qi ← [xi, T + ti];

0. for i ← 1, 2, ..., n do
       if cost-estimate(vi, Qi) = ∞ then
          fail;

   for each reserve, grab, or sequence directive do
       x ← the associated place node;
       t ← the time offset;
       if open?([x, T + t]) = false then
          fail;
       if this is a sequence directive then
          if T + t ≤ last_x then
             fail;

   if open?(Q0) = false or width(Q0) < width(v0) then
      fail;

1. if deliver-multiple([v1, v2, ..., vn], [Q1, Q2, ..., Qn]) = failure then
      fail;

2. for each reserve, grab, or sequence directive do
       x ← the associated place node;
       t ← the time offset;
       v ← a new value;
       if attach-label(v, [x, T + t]) = failure then
          fail;
       if this is a sequence directive then
          last_x ← T + t;

3. if attach-label(v0, Q0) = failure then
      fail;
   if make-live(v0) = failure then
      fail;
```

Figure 4.10: Scheduling x_0.t_0 `= f(`x_1.t_1`,` x_2.t_2`, ...,` x_n.t_n`)` at time T, with argument values $v_1, v_2, \ldots, v_n$. The result value is v_0. Here "fail" means undo everything and return.

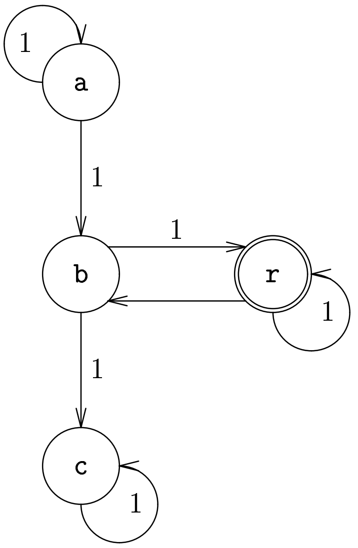

Figure 4.11: The place graph of Figure 4.5, before `r` is split to reflect the assignment of a variable to `r[0]`. The text assumes that there is a function microoperation that writes `a`, and one that reads `c`.

Safe nodes

The compiler defines a place-time node to be safe if it is of the form $[P, T_{\mathrm{safe}}]$ where P is a safe place node and T_{safe} is a particular time in the distant future. So that T_{safe} stays in the distant future as scheduling proceeds, the compiler increases T_{safe} whenever it labels a node $[P, T]$ such that $T_{\mathrm{safe}} - T$ is less than a somewhat arbitrary threshold.

A safe place node is a static node towards which a value can be moved without shrinking the set of place nodes to which it could potentially be delivered. A write-only register is an obvious example of an unsafe node. The precise definition, which follows, need not be understood to read the rest of the chapter.

I will illustrate the definition using the place graph of Figure 4.11. This describes the same architecture as the place graph of Figure 4.5, but the place node `r` has not yet been split into `r`$'$ and `r[0]` (which will be done because a variable is stored in `r[0]`). As I explained earlier, a transfer microoperation like the one that writes `r[0]` does not have a corresponding edge in the expanded graph, because it cannot be used in delivery routes. Such a microoperation cannot, however, be neglected in identifying the safe nodes, and in fact, the unexpanded graph is in general the right one to use for this purpose. In the example at hand, `r[0]` and `r`$'$ of Figure 4.5 will be safe if `r` of Figure 4.11 is safe.

In the place graph of Figure 4.11, there are four nodes: `a`, `b`, `c`, and `r`. Define the set *args* to include any place node that is read either by a function microoperation or by a transfer microoperation of the form `x[I] = y`. For the example, I will assume that the only node read by a function microoperation is `c`, so

$$args = \{\mathtt{b}, \mathtt{c}\}$$

Define the set *results* to include any place node that is written either by a function microoperation or by a transfer microoperation of the form `x = y[I]`, and also any place to which a variable will be assigned. In the example, I will assume that the only node written by a function microoperation is `a`, so

$$results = \{\mathtt{a}, \mathtt{b}, \mathtt{r}\}$$

Define $closure(x)$ to include any y such that there is a path from x to y in the place graph. If edges connect only nodes of the same width, a place node y is defined to be safe if

- y is static and not mutually exclusive with another node, and
- $\forall x \in results, \forall z \in args \cap closure(x),\;\; y \in closure(x) \Rightarrow z \in closure(y)$.

There are two safe nodes in the sample place graph:

$$safe = \{\mathtt{a}, \mathtt{r}\}$$

The reason that `c` is unsafe is that, although $\mathtt{a} \in results$ and $\mathtt{b} \in args \cap closure(\mathtt{a})$, and furthermore $\mathtt{c} \in closure(\mathtt{a})$, the right-hand side of the implication does not hold: $\mathtt{b} \notin closure(\mathtt{c})$.

After the RL compiler has identified the safe place nodes, it checks this reasonableness criterion:

$$\forall x \in results, \quad closure(x) \cap safe \neq \emptyset$$

It generates an error message for each x such that $closure(x) \cap safe = \emptyset$. Unfortunately, this criterion is somewhat unintuitive. I described a simpler criterion at the end of Section 3.3, which may be easier for architects to use, but is stronger than necessary.

In the general case, edges may connect place nodes of different widths. Define the width of a path to be the width of its narrowest node. Define $max\text{-}width(x, y)$ to be the width of the widest path from x to y, or zero if there is no path. In the general case, the place node y is defined to be safe if

- y is static and not mutually exclusive with another node, and
- $\forall x \in results, \forall z \in args \;\; \max\{max\text{-}width(x, y), max\text{-}width(x, z)\} \leq max\text{-}width(y, z)$.

The general reasonableness criterion is this: For all $x \in results$, there must be a path of monotonically non-decreasing width from x to a safe node.

Incremental updating

As nodes of the place-time graph are labeled, their effective capacities and the available spill paths change, but there is always required to be a consistent set of spill paths for the live values. The compiler must check this condition every time it labels a place-time node. Often the previous spill paths will still exist. To avoid a potentially prohibitive amount of unnecessary computation, the compiler does the checking incrementally. Specifically, to

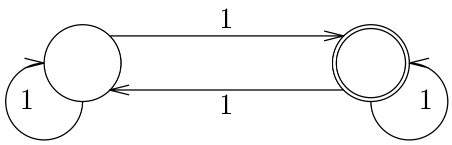

Figure 4.12: An extremely simple place graph. The node on the right will be assumed to have a capacity of two.

determine whether a consistent set of spill paths exists, the compiler actually tries to find such a set, by modifying the previous set that it found.

Figures 4.12 through 4.14 illustrate this. Figure 4.12 shows a place graph that is even simpler than the previous example. (Anything more complicated would become unwieldy later.) There are only two place nodes, which I will call `left` and `right`. To make things reasonably interesting, assume that `right` has a capacity of two. Finite capacities bigger than one come about when a pragma in the RL program or in the machine description is used to limit the number of registers in a register bank.

Figure 4.13 shows the corresponding place-time graph, with labels attached arbitrarily: value 1 on $[\texttt{left}, 0]$, and values 2 and 3 on $[\texttt{right}, 0]$. Spill paths for values 1–3 are shown, assuming that all three values are live. Figure 4.14 shows the adjusted spill paths, after a second node, $[\texttt{right}, 1]$, is labeled with value 1.

The labeling of a place-time node is only one of several different kinds of transactions that require updating of the set of spill paths:

- Labeling a node $[P, T]$ with a value v. The value v may be a dummy if P corresponds to a resource.

- Tagging a value v as live, or not live. The value v must already label some place-time node.

- Tagging a node $[P, T]$ as safe, or not safe. The compiler uses this to increase T_{safe}. For each $P \in \mathit{safe}$, it tags $[P, T_{\text{safe}} + 1]$ as safe and then tags $[P, T_{\text{safe}}]$ as not safe.

Transactions can fail. A successful transaction can be undone. This requires keeping an audit trail of the changes to the set of spill paths.[5]

Certain kinds of transactions can fail unpredictably: When labeling a place-time node, or when tagging a value as live, the spill-path updating algorithm described below may overlook a feasible way to adjust the set of spill paths. Little is required, however, for the code generator to make progress: One requirement—satisfied for any reasonable architecture—is that an attempt to label $[P, T]$ with v and then tag v as live will always succeed if T is large enough (such that $[P, T']$ is unlabeled for all $T' \geq T$).

[5]My compiler does not do this correctly; it uses heuristic spill-path updating to undo changes. It has never failed in practice, so I have not fixed it.

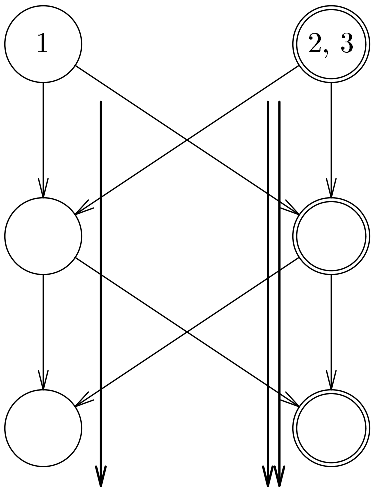

Figure 4.13: The place-time graph corresponding to Figure 4.12, with some labels and spill paths.

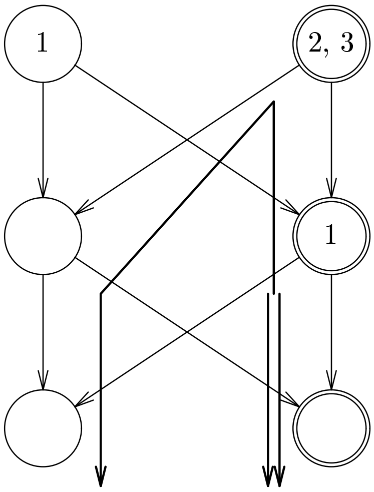

Figure 4.14: The result of adding another label, as if in delivering value 1 to the register bank on the right.

Delivery revisited

Constraining the labeling of nodes as required to maintain a set of spill paths reduces the likelihood of heading down a scheduling dead end, but avoiding such disaster altogether requires further refinement of the data routing algorithm. I will describe this here in a brief aside.

The spill path for a value is supposed to facilitate its delivery by providing a prefix for any delivery route. By keeping open a path through the early, heavily-labeled part of the place-time graph, one hopes to guarantee the existence of a delivery route to $[P, T]$ for all reasonable P and *some* T, but in fact, an open but tortuous route might not be discovered by the basic data routing algorithm of Figure 4.8. This problem can be solved with a backup delivery algorithm that makes explicit use of the known spill path of the value to be delivered. The complete procedure used in the compiler—shown in Figure 4.15—uses the basic algorithm as a subroutine. Indeed, the first thing it does is to try the basic algorithm, which tends to find a cheaper route when it does find a route.

Spill paths as network flow

Now I will describe the way in which the compiler is able to efficiently update the consistent set of spill paths. To label a place-time node, or to introduce a spill path for a new live value, it might well have to adjust every spill path in the set. Fortunately, there is a trick. The algorithm used in the compiler makes this adjustment in a worst-case time linear in the size of the place-time graph (truncated at T_{safe}). In the ideal case in which there are no mutually exclusive place-time nodes, the algorithm always finds a feasible set of spill paths if one exists. It may fail otherwise, but only infrequently if mutual exclusion is used sparingly.

The key idea behind the algorithm is to transform the problem of adjusting the set of spill paths in the place-time graph into a problem of adjusting a network flow on a different graph, the *place-time network*. Figure 4.16 shows the place-time network corresponding to the labeled place-time graph of Figure 4.13. It also shows the network flow corresponding to the spill paths. Unlike the place-time graph, the place-time network has capacity limits on its edges, not on its vertices. Certain edges in the network correspond to nodes in the place-time graph, and these have capacities equal to the effective capacities of the corresponding nodes.

Like the place-time graph, the place-time network is infinite. Not only does this fiction simplify this presentation; it is used within the compiler itself. Here is how the place-time network would be constructed in principle:

1. For each node Q in the place-time graph, create an *in-vertex* and an *out-vertex*, connected by an *internal edge* of capacity equal to the effective capacity of Q.

2. For each edge $Q_1 \to Q_2$ in the place-time graph, if $\mathit{width}(Q_1) \leq \mathit{width}(Q_2)$ connect the out-vertex of Q_1 to the in-vertex of Q_2 with an edge of unbounded capacity.

```
function deliver(v, Q) is

    current-mark ← a new mark;
    if deliver-aux(v, Q) = success then
        return success;

    if cost-estimate(v, Q) = ∞ then
        return failure;

    [Q_1, Q_2, ..., Q_n] ← spill-path-tail(v);
    if ∀i ∈ {2, 3, ..., n}, mark(Q_i) ≠ current-mark then
        return failure;

    make-dead(v);
    for i ← 2, 3, ..., n do
        attach-label(v, Q_i);  /* always succeeds */
    make-live(v);  /* always succeeds */

    current-mark ← a new mark;
    if deliver-aux(v, Q) = failure then
        undo the effects of this call to deliver;
        return failure;
    let Q'_1, Q'_2, ..., Q'_m be the delivery route;

    j ← 1;
    for i ← 2, 3, ..., n do
        if Q_i = Q'_1 then
            j ← i;
    for i ← j + 1, j + 2, ..., n do
        labels(Q_i) ← labels(Q_i) − {v};

    return success;

function spill-path-tail(v) is
    let Q_1, Q_2, ..., Q_k be the known spill path for v;
    j ← 1;
    for i ← 2, 3, ..., k do
        if v ∈ labels(Q_i) then
            j ← i;
    return [Q_j, Q_{j+1}, ..., Q_k];
```

Figure 4.15: The reliable data routing algorithm.

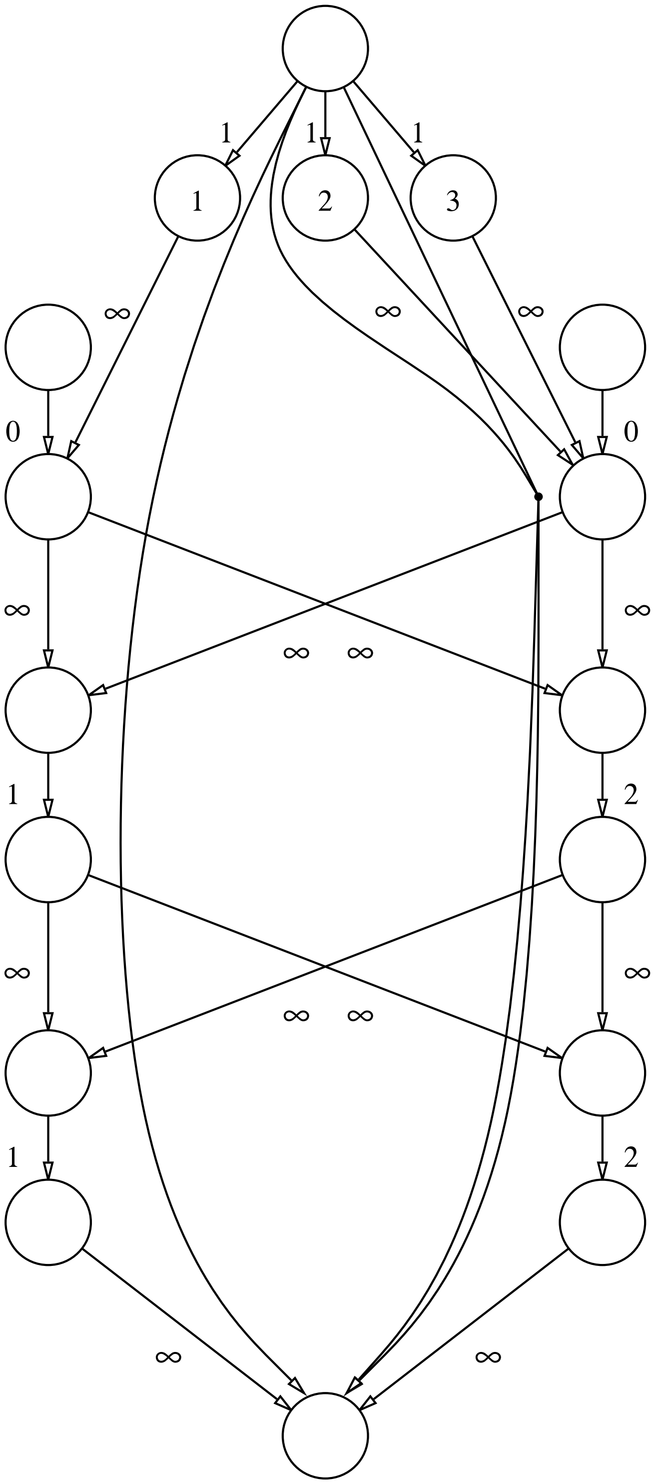

Figure 4.16: The place-time network and network flow corresponding to Figure 4.13.

3. For each value v, create a *value vertex*, and for each place-time node Q labeled by v, connect the value vertex of v to the out-vertex of Q with an edge of unbounded capacity.

4. Create a *source vertex* for the network. For each live value v, connect the source vertex to the value vertex of v with an edge of unit capacity.

5. Create a *sink vertex* for the network. For each safe place-time node Q, connect the out-vertex of Q to the sink vertex with an edge of unbounded capacity.

6. For each group of mutually exclusive place-time nodes, make the corresponding internal edges mutually exclusive. Call such a group of edges a *bundle*.

A flow through such a network is a function that assigns a non-negative integer, $\mathit{flow}(E)$, to each edge E, and satisfies these conditions:

- It is conserved at every vertex (but the source and the sink):

$$\sum_{\text{incoming } E} \mathit{flow}(E) = \sum_{\text{outgoing } E} \mathit{flow}(E)$$

- It satisfies the capacity constraints:

$$\mathit{flow}(E) \leq \mathit{capacity}(E)$$

- It accommodates every bundle of edges $E_1, E_2, \ldots, E_n$:

$$\sum_{i=1,2,\ldots,n} \mathit{flow}(E_i) \leq 1$$

If the net flow out of the source vertex equals the number of live values, then the network flow corresponds to a consistent set of spill paths for them. There is an ambiguity as to which spill path goes where when more than one unit of flow leaves an out-vertex, but the choice is indeed arbitrary.

Updating a network flow is a well-understood problem. Define an edge E to be *empty* if

$$\mathit{flow}(E) = 0$$

and *full* if

$$\mathit{flow}(E) = \mathit{capacity}(E)$$

An edge may well be neither empty nor full. Define an *augmenting path* to be an undirected path that traverses non-full edges forwards and non-empty edges backwards, and moreover, never traverses an edge of a bundle forwards unless either the other edges are empty or it traverses one of them backwards. To *augment* a network flow (by one unit) along an undirected path means to add a unit of flow to each edge that it traverses forwards, and to subtract a unit from each edge that it traverses backwards. Augmenting a network flow along an augmenting path preserves its correctness, except it causes a unit of flow to appear at the head of the path and disappear at the tail.

Figure 4.17 shows the use of an augmenting path to adjust the three spill paths in the example to accommodate another label, as in going from Figure 4.13 to Figure 4.14. The approach is to update the network—adding an edge from the value vertex of value 1 to the out-vertex of the labeled node, reducing the capacity of the internal edge from two to one to reflect the new effective capacity of the node, and thus forcing the reduction of the flow on that edge from two to one—creating temporary violations of flow conservation at the in-vertex and the out-vertex of the node. The procedure is then to try to find an augmenting path from the in-vertex to the out-vertex to correct the flow. This is the dashed, curved arrow in the figure. Augmenting the flow along it yields the properly adjusted flow shown in Figure 4.18.

An augmenting path is, of course, just a path through yet another graph, the *residual network* of network flow theory. If there are no bundles of mutually exclusive edges, then every path through the residual network is a valid augmenting path. In this happy case, if there is a way to cause an additional unit of flow to appear at one vertex and disappear at another, there is always an augmenting path that will accomplish this. Finding such an augmenting path involves nothing more than finding a path through the residual graph.

The case in which there are bundles is, in principle, hard. Determining whether a given network with bundles admits an integral flow of a given size is an NP-complete problem [19, problem ND36]. For such a network, augmenting the flow around a closed loop may cause an augmenting path to exist between a pair of vertices, between which none existed before. Moreover, just determining whether an augmenting path exists between two vertices is an NP-complete problem.[6] Fortunately, for realistic arrangements of bundles, augmenting paths exist and are easy to find for those operations in which finding a way to adjust the flow is necessary for the code generator to make progress.

The algorithm that the compiler uses to find an augmenting path resembles the basic data routing algorithm sans cost estimates. (Because the set of spill paths maintained by the compiler serves mainly as an existence proof—they are only used rarely in data routing—the quality of the augmenting path is not critical.) The compiler searches for an augmenting path with a depth-first search, locking and unlocking bundles of edges as it proceeds. Bundles that include non-empty edges are locked at the outset. The algorithm is shown in Figure 4.19. In the case in which there are no bundles, it is exact and, furthermore, optimal. In the general case, it is a heuristic that runs in the same worst-case time, without completely neglecting bundled edges. It never returns an augmenting path that puts more than one unit of flow on the edges of a bundle, and it occasionally finds an augmenting path that moves a unit of flow from one edge in a bundle to another.

Finally, here are the procedures that make incremental changes to the place-time network:

[6] It is NP-complete even if the flow on every edge is constrained to be zero [19, problem GT54].

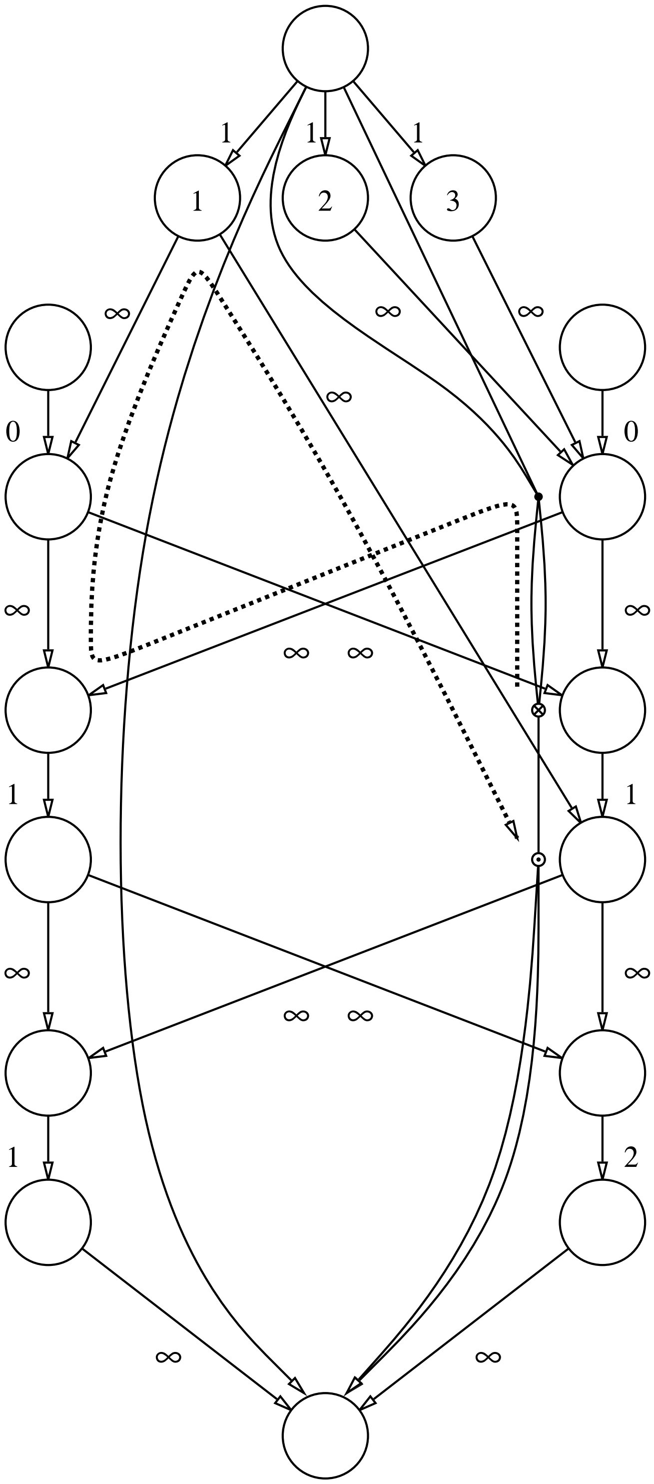

Figure 4.17: The network of Figure 4.16, updated to reflect the labeling of [`right`, 1] with value 1. The dashed, curved arrow is an appropriate augmenting path for correcting the resulting violations of flow conservation.

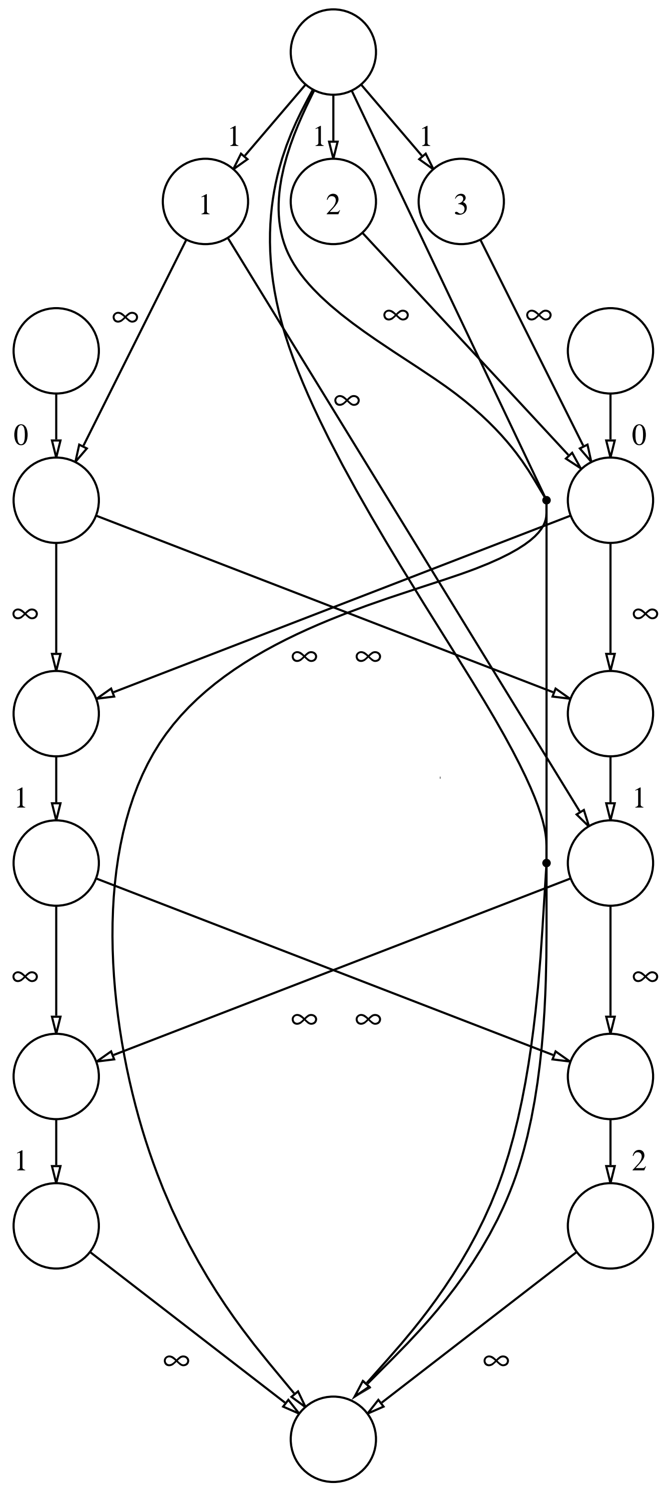

Figure 4.18: The adjusted network flow.

```
function augment(V_1, V_2) is
    current-vertex-mark ← a new mark;
    return augment-aux(V_1, V_2);

function augment-aux(V_1, V_2) is
    mark(V_1) ← current-vertex-mark;
    if V_1 = V_2 then
      return success;
    for each edge E to V_1 from some V do
        if mark(V) ≠ current-vertex-mark then
          if flow(E) > 0 then
            flow(E) ← flow(E) − 1;
            if augment-aux(V, V_2) = success then
              return success;
            flow(E) ← flow(E) + 1;
    for each edge E from V_1 to some V do
        if mark(V) ≠ current-vertex-mark then
          if flow(E) < capacity(E) and locked?(E) = false then
            flow(E) ← flow(E) + 1;
            if augment-aux(V, V_2) = success then
              return success;
            flow(E) ← flow(E) − 1;
    return failure;

function locked?(E) is
    if ∃E', flow(E') > 0 and E' is in a bundle with E then
      return true;
    else
      return false;
```

Figure 4.19: The algorithm for trying to augment the flow along some augmenting path from a vertex V_1 to a vertex V_2.

actually-attach-label(v, Q) : Put v into *labels*(Q), add an edge of unbounded capacity from the value vertex of v to the out-vertex of Q, and decrement the capacities of the internal edge of Q and any edges in the same bundle. If this leaves an edge E such that

$$\mathit{flow}(E) = \mathit{capacity}(E) + 1$$

then subtract one from *flow*(E) and try to augment the flow along some augmenting path from the in-vertex of E to the out-vertex of E.

make-live(v) : Add an edge of unit capacity from the source to the value vertex of v, and then try to augment the flow along some augmenting path from the source to the sink.

make-dead(v) : Remove the edge from the source to the value vertex of v, and then rip up one unit of flow along some path from the value vertex of v to the sink.

make-safe(Q) : Add an edge of unbounded capacity from the out-vertex of Q to the sink.

make-unsafe(Q) : Remove the edge from the out-vertex of Q to the sink, and then for each unit of flow that it carried, augment the flow along some augmenting path from the out-vertex of Q to the sink.

Chapter 5

Evaluation

The RL compiler has been used by at least a half dozen different people here at Berkeley. It is written in Common Lisp, and it compiles typical programs, for Kappa, at a rate of about one instruction per second, which is fast enough.[1] There are two areas in which I want to carefully evaluate exactly how well the compiler does: the quality of the code generated by the algorithms of Chapter 4, and the usefulness of the compiler in developing application-specific architectures.

5.1 Efficiency of generated code

The function of the algorithms of Chapter 4—top-level scheduling, data routing, and spill-path updating—is to generate a straight-line sequence of instructions from a DAG. I will show that they succeed in generating sequences that are tolerably close in length to hand-written sequences, provided extremely short sequences are excluded. The proviso is necessary because the search that the compiler conducts is sharply limited. There can be no guarantee that it will find a tricky one-instruction solution, and doubling the running time of an inner loop is not tolerable.

Global optimizations—software pipelining or removal of loop-invariant computations, for instance—may be critical for some programs, but such programs are not useful for evaluating the local algorithms of Chapter 4. Fortunately, they have been the exception in our experience with the RL compiler. When they do come up, it is often possible for the programmer to work around the problem; partial loop unrolling can substitute for software pipelining, for instance.

My strategy will be to compare compiler-generated code, for real RL programs, with functionally equivalent hand-written code. Table 5.1 shows instruction counts for hand-written and RL versions of three sample programs, all for Kappa: `controller`, `pitch`, and `xmas`. As can be seen from the static instruction counts in the table, these are medium-sized

[1] Spill path updating uses a significant fraction of the compilation time, but surprisingly, not as much as the data-flow cost estimation of Section 4.2.

program	hand-written		RL		increase	
	static	dynamic	static	dynamic	static	dynamic
`controller`	175	646	268	657	53%	2%
`pitch`	93	1921	101	1926	9%	0%
`xmas`	119	104	142	123	19%	18%

Table 5.1: Instruction counts for hand-written and RL versions of three programs.

programs; the RL versions consist of 100–200 lines each. The RL version of `pitch` and the corresponding compiler-generated code are included in the appendices.

Since I am more familiar with the RL compiler, and perhaps with Kappa, than anyone else can be expected to be, ideally I should not be the author of either version of any of the sample programs. Unfortunately, I had to have a hand in the RL versions of `controller` and `pitch`, and in the hand-coded version of `xmas`. I did my best to exercise restraint so that the comparison would nevertheless be fairly realistic.

Each sample program has an initialization section and a main loop that repeats indefinitely. Table 5.1 lists, for each version of each program, a static and a dynamic instruction count. The static count is just the total number of instructions in the program. The dynamic count is the number of instructions executed per iteration of the main loop in the worst case, assuming that all polling loops execute just once.[2]

Controller

This is a two-axis adaptive robot controller designed by Azim [7,6]. The hand-written code appears in his thesis, along with a partial implementation in C. To obtain a complete RL version, I decompiled the hand-written code, styled the result to resemble the existing C code, and passed it to Azim for criticism.

The instruction counts show that the compiled RL version runs at about the same speed as the hand-written version, but is about fifty percent bigger. Azim worked hard to keep his code small, defining small subroutines and loops as one would not do in RL. Table 5.2 shows that the hand-written version indeed executes, per iteration of the main loop, fewer blocks (straight-line code segments) than the RL version. This accounts for the larger size of the RL code. It also accounts for the apparently unreasonably-good code generated by the compiler; the tradeoff of speed for space in the hand-written code balances the poorer quality of the compiler-generated code.

In interpreting the instruction counts, it is important to be aware that `controller` spends about seventy percent of its time in a subroutine for multiplication, as is apparent from Table 5.3. Each multiplication of two 20-bit variables takes 20 instruction cycles

[2] In `controller`, there is a loop whose repetition count is externally determined. I have assumed that its value is one, which the author, Azim, has told me is typical.

program	hand-written		RL	
	static	dynamic	static	dynamic
`controller`	36	87	41	56
`pitch`	6	55	10	139
`xmas`	6	3	9	5

Table 5.2: Block counts for the programs.

program	hand-written		RL	
	static	dynamic	static	dynamic
`controller`	11%	72%	7%	70%
`pitch`	0	0	0	0
`xmas`	17%	19%	14%	16%

Table 5.3: Fractions of the instruction counts due to two-variable multiplication.

(and returning from the subroutine takes another cycle). The other programs, `pitch` and `xmas`, do more multiplications by constants, for which the compiler generates much shorter instruction sequences.

Pitch

This is a variant of the Gold pitch-tracking algorithm [20], which has been used as a test case in the Lager project for a long time. It detects peaks and valleys in an input signal, and using a moderately complex voting scheme, makes estimates of the pitch for use in voice encoding. This particular hand-coded version appears in the thesis of Shung [38]. I updated an old RL implementation of the algorithm to make it functionally equivalent to the hand-written code, using a decompiled version of the hand-written code as a guide.

The compiled RL version runs about as fast as the hand-written version, and is about ten percent bigger. The overhead in the speed of the compiler-generated code is again disguised. In this case, however, the compiler-generated version executes more blocks per iteration of the main loop, not fewer. The cause is a heavy reliance, in the hand-written code, on conditional microoperations. The corresponding RL code is, naturally, written with `if` statements, for which the compiler generates branches. The RL compiler's approach produces faster code (for this particular program).

Xmas

This is the program for a demonstration chip designed by Richards and Thaler [34]. The name derives from the December 1988 deadline for the completion of the initial design. The function of the program is simply to detect the presence of sound in a specific frequency range. The biggest part of it is a single straight-line segment, consisting of a lot of carefully designed arithmetic that does the signal processing. Richards and Thaler wrote only an RL version of the program, which I compiled by hand into near-optimal code. The compiler-generated version is about twenty percent slower and about twenty percent bigger than my hand-written version.

Conclusion

I do not promise that the RL compiler will generate such good code for all programs, but certainly these examples demonstrate that the algorithms of Chapter 4 do indeed produce good straight-line code. They also demonstrate that real programs that do not have one-instruction inner loops are plentiful. The average block sizes in the RL versions of the three sample programs are 6.5, 10, and 16 instructions, and the worst-case dynamic instruction and block counts suggest that the frequency-weighted average block sizes are no smaller. Moreover, in `controller`, which uses two-variable multiplication heavily, the average length of the schedule generated from a DAG is significantly larger than the average size of a block; a single DAG may generate many blocks, interspersed with calls to the multiplication subroutine.

5.2 Case studies in retargeting

To be useful in experimenting with designs for application-specific architectures, the RL compiler must be easy to retarget. I have experimented with a variety of diverse machine descriptions, but these were really just exercises. Here I want to describe two case studies in which the compiler is actually being used to design application-specific chips based on modified versions of the Kappa architecture. In the second of these, it is the chip designer, not the compiler writer, who has done the retargeting. Unlike the three examples of the previous section, the RL programs involved here are many hundreds of lines long, and there are no plans to produce hand-coded versions for comparison.

Extending Kappa for a robotics application

The first case study concerns the incorporation of additional hardware into Kappa by Thon. In his research on computer-aided design [43], Thon is developing a chip for performing a computationally intensive task normally assigned to the host processor of a robot arm: solving for sets of joint angles that correspond to a desired position and orientation of the manipulator.

The program compiles into just under 700 instructions—a tight fit into on-chip read-only memory. By compiling it for Kappa, Thon confirmed that additional hardware is needed to achieve the required speed. The program spent too much time in the multiplication routine, and too much time doing variable-distance shifts by repeatedly shifting by one bit at a time. The variable-distance shifts occur in cordic computations of sine, arctangent, and square root.

Figure 5.1 shows Thon's extended version of Kappa. It incorporates a parallel multiplier and a new, logarithmic shifter, which can shift by fifteen bits. Thon's architecture allows the shift distance to be obtained from within the address unit so that the cordic routines can use `x >> k`, and store `k`, a loop variable, in the address unit.

The differences between the machine descriptions for Kappa and for the extended architecture are small. The hardware itself is another matter; in fact the parallel multiplier has only recently been fabricated and tested. Thon will first fabricate a version of his chip that uses serial multiplication. Working with multiple architectures in this manner would be difficult without a retargetable compiler for a fairly machine-independent language.

A processor for mobile radio

The second case study involves more extensive modification of Kappa. Svensson is investigating tradeoffs—in architecture and in adaptive filtering algorithm—for an application-specific chip that is to perform channel equalization for a digital mobile telephone [42]. In this case, the designer has taken responsibility for maintaining the machine description, and has written a substantial part of it from scratch.

A typical version of the program compiles into about 400 instructions, about thirty percent of which are due to aggressive loop unrolling in the source program. The least demanding version has to perform more than 2.5 million multiplications per second, in addition to other operations. Svensson's goal in modifying the architecture has been to achieve this with a processor that will execute, conservatively, about 5 million instructions per second. Because the requirements will differ according to the algorithm and the clock rate, he ultimately intends to define a family of architectures, providing a range of performance.

Figure 5.2 shows the main arithmetic unit of Svensson's current top-of-the-line architecture. He began with Thon's architecture (without the fancy shifter), and looked for ways to improve the performance. Since the bandwidth from memory was insufficient to keep the multiplier and adder busy, he decided to use the two independent memories shown in the figure. Comparison of this figure with the previous will reveal additional, minor changes. (The reason that there is no output port is that the only output is a bit stream generated by the boolean unit.)

The introduction of the second memory was originally a troublesome architectural change. It influenced the incorporation of register type modifiers into RL. Now the language and compiler should be more compatible with architectural features of this particular nature. Svensson originally planned to generate addresses for the pair of memories using a pair of address units identical to Kappa's. He later designed a single unified address unit.

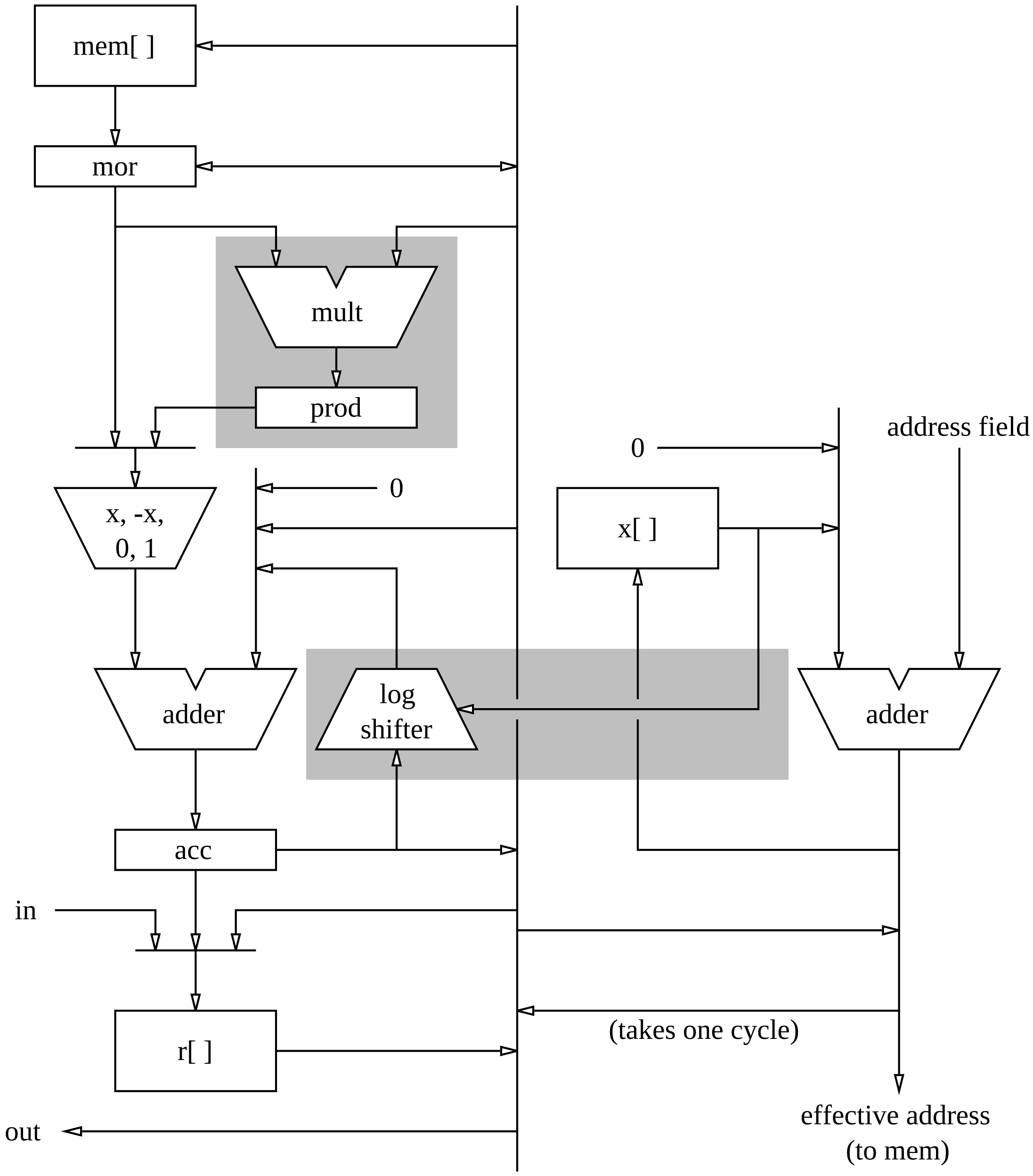

Figure 5.1: Thon's extended Kappa datapath.

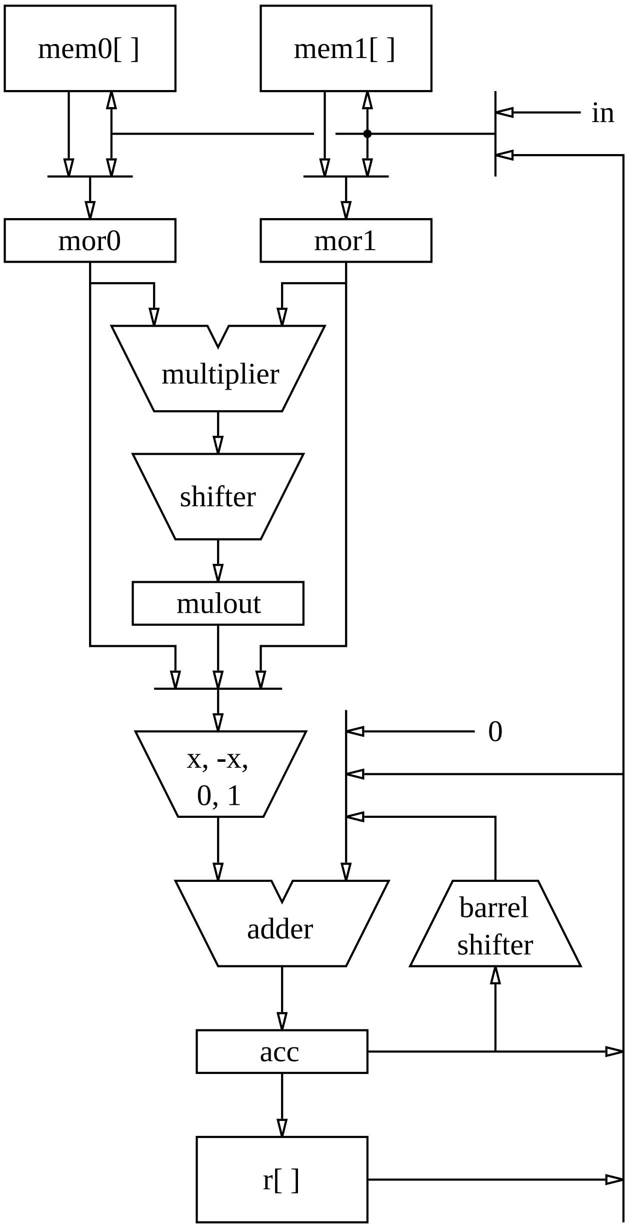

Figure 5.2: Svensson's fixed-point datapath.

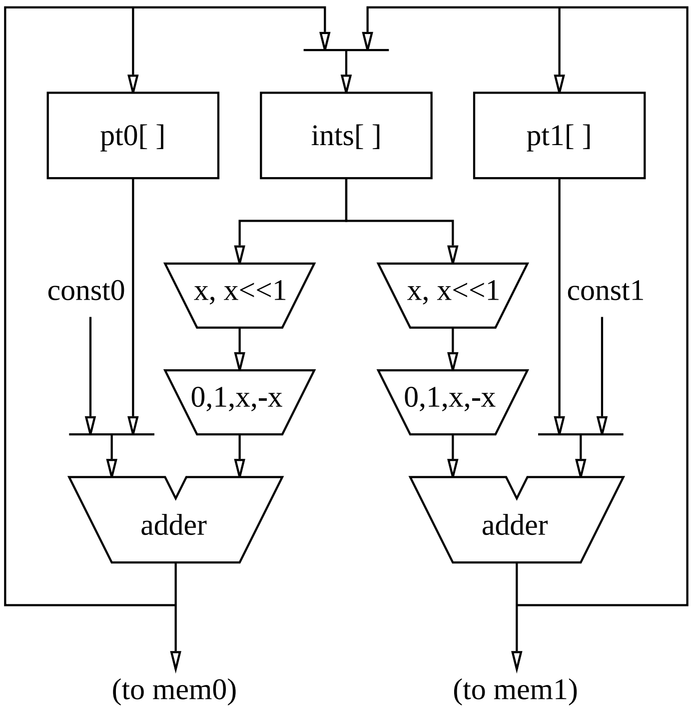

Figure 5.3: Svensson's integer and pointer datapath.

For this there were no modifications necessary to the compiler, and furthermore, I was not involved in modifying the machine description.

Figure 5.3 shows the new address unit. The design is essentially an example of architectural support for the RL programming language. With the old approach of using two address units, to efficiently compute the inner product of a vector stored in one memory with a vector stored in the other, the programmer had to use two loop variables stored in the corresponding address units. (The RL compiler does no global optimization and is incapable of deciding to use two variables where the programmer specified one.) With the new address unit, the programmer can index both vectors with a single loop variable, an `int`. The names of the three register banks—`int`, `pt0`, and `pt1`—are suggestive of their intended uses. In the machine description, Svensson directs the compiler to, by default, use these register banks to store `int` variables, pointers into `mem0`, and pointers into `mem1`, respectively.

Svensson sketched for several days before settling on the basic design of the address unit, but writing and debugging a description of it took only a day. He simplified the design over the course of another few days, finally obtaining the result shown in the figure. There were two main instances in which the compiler caused difficulties.

First, it occasionally performed part of an address calculation in the main arithmetic unit, causing time to be wasted later in transferring the result back to the address unit.

Since Svensson had designed the address unit to be self-sufficient, he solved this problem by just removing the connection between the units, as he would have done eventually anyway. I could improve the compiler's performance in such cases by using delivery cost estimates, instead of the order of definition in the machine description, to break ties between different microoperations for implementing the same operation, but I am reluctant to do this without an example of a real case in which it matters.

Second, the machine description did not follow my recommendation (in Section 3.3) against using macros or unnecessary compiler directives in defining transfer microoperations. Svensson described the transfer microoperations that pass data through each of the adders in the address unit as macros, each expanding into two microoperations: the first to set the other input of the adder to zero, and the second to add. The microoperations so defined in effect invoke the `reserve` compiler directive on the input that is set to zero. The nature of the architecture, however, is such that no microoperation that writes that input would ever be scheduled in the same instruction anyway. We changed the machine description so that it does not define the transfer microoperations as macros. For one program, the change caused the compiler to generate 387 instructions instead of 390. It is unfortunate that the compiler is sensitive in this way, but in this case at least, it would have been okay to ignore the issue.

All in all, the compiler has worked well enough for Svensson to be able to concentrate on algorithmic and architectural issues. It has enabled him to put algorithm and architecture together at an early stage in the design process, so that each can be developed with realistic assumptions about the other. It has also enabled him to work with a variety of algorithms and a variety of architectures. This compiler-based design methodology will be one of the topics of his dissertation, as will be an evaluation of the advantages of using different processors for different adaptive filtering algorithms [41].

Chapter 6

Conclusion

I have described a family of programmable processors and a user-retargetable compiler that form the basis of a practical development strategy for application-specific integrated circuits. In this strategy, which is applicable, for example, to audio signal processing applications, the designer first writes a program, and then develops a suitable processor architecture by incrementally modifying a preexisting architecture, observing the effect of each change on the compiler-generated code. I have described two cases in which my collaborators are using this strategy to design complex chips. Our experience so far indicates that the machine description language of the RL compiler is indeed flexible enough and simple enough for the compiler to be used to assess architectural changes. The compiler seems to be more reliable than intuition for this purpose.

The target processors, typified by Kappa, use horizontal instructions and datapaths of irregular topology. The choice of this architectural style facilitates the customization of a processor to suit a particular program, but limits the applicability of standard code-generation techniques. I have shown that the combination of greedy scheduling and lazy data routing used in the RL compiler produces straight-line code for these processors that is competitive with hand-written code. Moreover, retargeting of the compiler seldom requires writing code—I attribute this to the ability of lazy data routing to utilize diverse datapath topologies.

In datapaths that have hot spots, as ours do, lazy data routing is complicated by the possibility of making a bad decision that precludes ever using a particular value. The way to do lazy data routing without extensive backtracking is to test for the existence of *spill paths*. I have described an incremental algorithm for this that is based on finding augmenting paths for network flows. The algorithm is a heuristic that works best when the instruction format is fully horizontal.

I will say a little about the direction in which this research should go next. Adding complexity to the compiler is not the way to go. I would neither incorporate more optimizations, nor try to extend the class of target architectures. In the big picture, what I have done in this thesis is describe a combination of an architectural family and a compiler. A deficiency of this particular combination is that the code generator is fairly complex, since it must test for the existence of spill paths. Another deficiency is that the class of target architectures—although it is suitably diverse and open-ended—is difficult to define precisely. What I would do next is design one or two other combinations of architectural family and compiler to see how they compare. The current combination, however, is tough competition: Implementations of Kappa and its relatives could, in principle, be made quite small and fast. We still have much to learn about high-level-language-oriented architecture for digital signal processors in general, and for application-specific processors in particular.

Appendix A

The Kappa machine description

```
architecture "Vanilla Kappa"

#define bus     node : delay = 0
#define latch   node : delay = 1
#define reg     node : static : delay = 1
#define file    node : static : delay = 1 : bank
#define memory  node : static : delay = 1 : bank : symbolic_addresses


/* Defaults */

pragma word_length 12
pragma max_left_shift 1
pragma max_right_shift 6

pragma fix_register r
pragma int_register x
pragma bool_register bmem
pragma mem_pointer_register x

pragma fix_memory mem
pragma int_memory mem
pragma bool_memory bmem
pragma mem_pointer_memory mem
```

```
/* Arithmetic Unit */

reg mor
bus mbus, abus, bbus, sum
latch acc
bus sign, rbus
reg cc
file r

resource sat_abus  /* Don't combine sat minus with nonsat addition. */
resource sat_sum   /* Don't combine nonsat minus with sat addition. */

micro   mor = mbus
micro   mbus = mor

micro   abus = 0
micro   abus = 1
micro   abus = mor
micro   abus = -mor
                grab sat_sum
micro   abus = abs(mor)
                grab sat_sum
micro   abus = sat(-mor)
                grab sat_abus
micro   abus = sat(abs(mor))
                grab sat_abus

micro   bbus = 0
micro   bbus = mbus
micro   bbus = acc
micro   bbus = acc >> N
micro   bbus = acc << N
```

```
micro   sum = abus
micro   sum = bbus
micro   sum = abus + bbus
                grab sat_abus
micro   sum = bbus + abus
                grab sat_abus
macro   sum = bbus - mor
                { abus = -mor, sum = abus + bbus }
micro   sum = sat(abus + bbus)
                grab sat_sum
micro   sum = sat(bbus + abus)
                grab sat_sum
macro   sum = sat(bbus - mor)
                { abus = sat(-mor), sum = sat(abus + bbus) }
micro   sign = (sum.-1 < 0)
micro   acc = sum
micro   acc = cc ? sum : acc

micro   mbus = acc
micro   rbus = acc
micro   rbus = mbus
micro   r[N] = rbus
micro   mbus = r[N]

resource input_output

micro   rbus = in(Port)
                sequence input_output
micro   out(mbus, Port)
                sequence input_output

macro   bbus.1 = (abus + bbus) >> N    /* N > 0! */
                { sum = abus + bbus, acc = sum;
                  bbus = acc >> N; }
macro   bbus.1 = (bbus + abus) >> N    /* N > 0! */
                { sum = bbus + abus, acc = sum;
                  bbus = acc >> N; }
macro   bbus.1 = (bbus - mor) >> N    /* N > 0! */
                { abus = -mor, sum = abus + bbus, acc = sum;
                  bbus = acc >> N; }
macro   bbus.1 = -mor >> N            /* N > 0! */
                { abus = -mor, sum = abus, acc = sum;
                  bbus = acc >> N; }
```

```
/* Address Unit */

file x
bus addr, xbus, xsum, xsign, eabus

micro   addr = Immediate
micro   xbus = x[N]
micro   xsum = addr
micro   xsum = xbus
micro   xsum = addr + xbus
micro   xsum = xbus + addr
micro   xsign = xsum.-1 < 0
micro   eabus = xsum
micro   x[N] = eabus


/* Memory */

memory mem

resource read_write
micro   mor = mem[eabus]
                grab read_write
micro   mem[eabus] = mbus
                grab read_write

macro   mor = mem[Loc]
                { addr = Loc, xsum = addr, eabus = xsum, mor = mem[eabus] }
macro   mem[Loc] = mbus
                { addr = Loc, xsum = addr, eabus = xsum, mem[eabus] = mbus}

macro   mor = mem[Loc + xbus]
                { addr = Loc, xsum = addr + xbus, eabus = xsum,
                              mor = mem[eabus] }
macro   mem[Loc + xbus] = mbus
                { addr = Loc, xsum = addr + xbus, eabus = xsum,
                              mem[eabus] = mbus }
```

```
/* Multiplication */

resource multiplication

micro   load_coef(mbus)
                sequence multiplication

micro   bbus.1 = multiply_start(mor)
                reserve abus, sum, acc.1
                sequence multiplication

micro   bbus.1 = multiply_step(mor, bbus)
                reserve abus, sum, acc.1
                sequence multiplication

micro   acc = multiply_finish(mor, bbus)
                reserve abus, sum
                sequence multiplication

micro   acc.0 = multiply_subroutine(mor)
                reserve mbus, sign, xsign
                sequence multiplication


/* Miscellaneous */

resource one_way_only

micro   eabus = mbus
                grab one_way_only
                unless _address_word_length
micro   mbus.1 = eabus
                grab one_way_only
                unless _address_word_length

op "write_timer_register" write_timer_register(x)
micro   write_timer_register(eabus)
                /* a non-standard microoperation type */
```

```
/* Boolean Unit */

bus:bank bval
memory bmem

micro   bval[N] = sign
micro   bval[N] = xsign
micro   bval[N] = bmem[Loc]
micro   bval[N] = cc

micro   cc = bval[N]
micro   bmem[Loc] = bval[N]

micro   bval[I] = true()
micro   bval[I] = false()
micro   bval[I] = !bval[J]
micro   bval[I] = bval[J] && bval[K]
micro   bval[I] = bval[J] || bval[K]
```

Appendix B

A Sample RL Program

B.1 The RL version of pitch

```
/*
 *  Gold Pitch Tracker
 */

#pragma arch_file "vanilla-kappa"
#pragma word_length 16
#pragma r_capacity 2
#pragma x_capacity 3
#pragma _max_sample_interval 350

#define VOICED 5
#define BLANK 12
#define DECAY (1 - 3/128)
#define DELTA 4

#define compare(a, b) (abs((a) - (b)) < DELTA)
#define is_peak(x, old_x) ((old_x) && !(x))
#define is_valley(x, old_x) (!(old_x) && (x))

fix sig, old_sig, last_peak, last_valley, signal[6], thresh[6];
int score, topscore, pitch, winner, ppc[6], pp[6], old_pp[6];
bool slope, old_slope, even;
const volatile bool EOS;

fix low_pass(x)         /* transfer function = -1/4 / (1 - 3/4 z^-1) ^ 2 */
     fix x;
{
```

```
  static fix y, z;

  y += (1/4) * (x - y);
  z = (3/4) * z - y;
  return z;
}

int tally_score(score, a, b, c)
     int score, a, b, c;
{
  score = compare(a, b) ? score + 1 : score;
  score = compare(a, c) ? score + 1 : score;
  score = compare(a, b + c) ? score + 1 : score;
  return score;
}

void init() {
  write_timer_register((int) 350);
}

void loop() {
  register int i;

  pitch = (topscore < VOICED) ? 0 : winner;
  topscore = 0;

  for (i = 0; i < 6; i++) {
    register int j;

    out(pitch);

    old_sig = sig;
    sig = low_pass(in());
    old_slope = slope;
    slope = (sig >= old_sig);

    signal[0] = sig;
    signal[1] = -signal[0];
    signal[2] = sig/2 - last_valley/2;
    signal[3] = -signal[2];
    signal[4] = sig/2 - last_peak/2;
    signal[5] = -signal[4];
```

```
    last_valley = is_valley(slope, old_slope) ? sig : last_valley;
    last_peak = is_peak(slope, old_slope) ? sig : last_peak;

    score = 0;
    even = 1;

    for (j = 0; j < 6; j++) {
      fix threshold;
      bool after_blank, is_extremum;

      after_blank = ++ppc[j] >= BLANK;
      is_extremum = even && is_peak(slope, old_slope)
        || !even && is_valley(slope, old_slope);
      even = !even;

      threshold = thresh[j];
      threshold = after_blank ? DECAY * threshold : threshold;
      thresh[j] = threshold;

      if (signal[j] > threshold && after_blank && is_extremum) {
        thresh[j] = signal[j];
        old_pp[j] = pp[j];
        pp[j] = ppc[j];
        ppc[j] = 0;
      } else {
        old_pp[j] = old_pp[j];
        pp[j] = pp[j];
      }
      score = tally_score(score, pp[i], pp[j], old_pp[j]);
    }

    do {
    } while (!EOS);

    if (score >= topscore) {
      winner = pp[i];
      topscore = score;
    }
  }
}
```

B.2 The compiler-generated code

```
/*  "pitch.k" compiled for "Vanilla Kappa".  */

.PARAMETERS

arch_file = "vanilla-kappa";
word_length = 16;
stack_depth = 0;
start_state = 0;
max_sample_interval = 350;

.DATA

const_volatile_bool EOS;
x       3;
r       2;
mem     sig, last_peak, last_valley, signal[6], thresh[6], score,
        topscore, pitch, winner, ppc[6], pp[6], old_pp[6], y, z,
        _MEM_TEMP[2];
bmem    slope, old_slope, even, EOS, _BC, _BMEM_TEMP, _BMEM_TEMP_1;

.CODE   /* 11 blocks with a total of 101 instructions */

0:      /* 1 instructions */

   /*
    *   write_timer_register((int ) 350);
    */

    eabus=xsum=addr=350, write_timer_register(eabus);

                    GOTO 1;

1:      /* 6 instructions */

   /*
    * pitch.k, 51: pitch = topscore + (int ) -5 < 0 ? (int ) 0 : winner;
    * pitch.k, 52: topscore = (int ) 0;
    *              i = (int ) 0;
    */

    eabus=xsum=addr=&topscore, mor=mem[eabus];
    mbus.1=eabus=xsum=addr=-5, acc=sum=abus=0;
    bbus=mbus, abus=mor, eabus=xsum=addr=&winner, mor=mem[eabus],
        r[0]=rbus=acc, sum=abus+bbus;
    acc=sum=abus=mor, mem[eabus]=mbus=r[0], eabus=xsum=addr=&topscore,
        sign=sum.-1<0, CC:=SIGN;
```

```
    sum=abus=0, x[0]=eabus=mbus=r[0], acc=cc?sum:acc;
    mem[eabus]=mbus=acc, eabus=xsum=addr=&pitch;

                    GOTO 2;

2:      /* 30 instructions */

   /*
    * pitch.k, 57: out(pitch, 0);
    * pitch.k, 59: old_sig = sig;
    *              x = (fix ) in(0);
    * pitch.k, 30: y = y + (fix ) 0.25 * (x + - y);
    * pitch.k, 31: z = (fix ) 0.75 * z + - y;
    *              low_pass_result = z;
    * pitch.k, 60: sig = low_pass_result;
    * pitch.k, 61: old_slope = slope;
    * pitch.k, 62: slope = ! (sig + - old_sig < 0);
    * pitch.k, 64: (&signal)[(int ) 0] = sig;
    * pitch.k, 65: (&signal)[(int ) 1] = - (&signal)[(int ) 0];
    * pitch.k, 66: (&signal)[(int ) 2] = (fix)0.5*sig + (fix)-0.5*last_valley;
    * pitch.k, 67: (&signal)[(int ) 3] = - (&signal)[(int ) 2];
    * pitch.k, 68: (&signal)[(int ) 4] = (fix)0.5*sig + (fix)-0.5*last_peak;
    * pitch.k, 69: (&signal)[(int ) 5] = - (&signal)[(int ) 4];
    * pitch.k, 71: last_valley = ! old_slope && slope ? sig : last_valley;
    * pitch.k, 72: last_peak = old_slope && ! slope ? sig : last_peak;
    * pitch.k, 74: score = (int ) 0;
    * pitch.k, 75: even = (bool ) 1;
    *              j = (int ) 0;
    */

    eabus=xsum=addr=&pitch, mor=mem[eabus], _BMEM_TEMP:=!slope,
        even:=TRUE(), old_slope:=slope;
    mbus=mor, eabus=xsum=addr=&y, mor=mem[eabus], out(mbus,0),
        _BMEM_TEMP_1:=old_slope;
    eabus=xsum=addr=&sig, mor=mem[eabus], acc=sum=abus=sat(-mor),
        r[0]=rbus=in(0);
    r[0]=rbus=acc, mor=mbus=r[0], x[1]=eabus=xsum=addr=0,
        acc=sum=abus=sat(-mor);
    bbus=mbus=r[0], abus=mor, eabus=xsum=addr=&y, mor=mem[eabus],
        r[0]=rbus=acc, acc=sum=sat(abus+bbus);
    abus=mor, eabus=xsum=addr=&z, mor=mem[eabus],
        acc=sum=sat(abus+bbus), bbus=acc>>2;
    mem[eabus]=mbus=acc, acc=sum=abus=mor, r[1]=rbus=acc,
        eabus=xsum=addr=&y;
    abus=mor, acc=sum=bbus+abus, bbus=acc>>1;
    mor=mbus=r[1], acc=sum=bbus=acc>>1;
    bbus=acc, mor=mbus=r[0], acc=sum=sat(bbus+abus), abus=sat(-mor);
    mor=mbus=acc, r[0]=rbus=acc, abus=mor, bbus=acc,
```

```
        sum=sat(bbus+abus), eabus=xsum=addr=&z, mem[eabus]=mbus;
    mem[eabus]=mbus=r[0], acc=sum=abus=sat(-mor), sign=sum.-1<0,
        eabus=xsum=addr=&sig, _BMEM_TEMP:=_BMEM_TEMP&&!SIGN,
        _BMEM_TEMP_1:=SIGN&&_BMEM_TEMP_1, slope:=!SIGN;
    acc=sum=bbus=mbus=r[0], r[0]=rbus=acc, eabus=xsum=addr=&signal,
        mem[eabus]=mbus;
    mem[eabus]=mbus=r[0], r[0]=rbus=acc, acc=sum=bbus=acc>>1,
        eabus=xsum=addr=&signal+1;
    eabus=xsum=addr=&last_valley, mor=mem[eabus], r[1]=rbus=acc,
        acc=sum=abus=0;
    mor=mbus=r[1], r[1]=rbus=acc, acc=sum=abus=-mor;
    abus=mor, mem[eabus]=mbus=r[1], eabus=xsum=addr=&score,
        acc=sum=sat(abus+bbus), bbus=acc>>1;
    mor=mbus=acc, eabus=xsum=addr=&signal+2, mem[eabus]=mbus;
    mor=mbus=r[0], acc=sum=abus=sat(-mor);
    mem[eabus]=mbus=acc, acc=sum=abus=mor, eabus=xsum=addr=&signal+3;
    eabus=xsum=addr=&last_peak, mor=mem[eabus], r[0]=rbus=acc,
        acc=sum=bbus=acc>>1;
    mor=mbus=acc, acc=sum=abus=-mor;
    abus=mor, acc=sum=sat(abus+bbus), bbus=acc>>1;
    mor=mbus=acc, eabus=xsum=addr=&signal+4, mem[eabus]=mbus;
    eabus=xsum=addr=&last_valley, mor=mem[eabus],
        acc=sum=abus=sat(-mor);
    mem[eabus]=mbus=acc, acc=sum=abus=mor, eabus=xsum=addr=&signal+5,
        CC:=_BMEM_TEMP;
    sum=bbus=mbus=r[0], eabus=xsum=addr=&last_peak, mor=mem[eabus],
        acc=cc?sum:acc;
    mem[eabus]=mbus=acc, acc=sum=abus=mor,
        eabus=xsum=addr=&last_valley, CC:=_BMEM_TEMP_1;
    sum=bbus=mbus=r[0], acc=cc?sum:acc;
    mem[eabus]=mbus=acc, eabus=xsum=addr=&last_peak;

                    GOTO 3;

3:      /* 13 instructions */

   /*
    * pitch.k, 81: after_blank = !(((&ppc)[j]=(&ppc)[j]+(int)1)+(int)-12<0);
    * pitch.k, 83: is_extremum = even && (old_slope && !slope)
                              || !even && (!old_slope && slope);
    * pitch.k, 84: even = ! even;
    * pitch.k, 86: threshold = (&thresh)[j];
    * pitch.k, 87: threshold = after_blank?(fix)0.9765625*threshold:threshold;
    * pitch.k, 88: (&thresh)[j] = threshold;
    *              _BC = (threshold+-(&signal)[j]<0&&after_blank)&&is_extremum;
    */

    xbus=x[1], acc=sum=abus=1, addr=&ppc, eabus=xsum=addr+xbus,
```

```
        mor=mem[eabus],
        _BMEM_TEMP:=AND(even,old_slope,!slope)||AND(slope,!even,!old_slope),
        even:=!even;
    bbus=acc, abus=mor, xbus=x[1], addr=&thresh, eabus=xsum=addr+xbus,
        mor=mem[eabus], acc=sum=abus+bbus;
    xbus=x[1], mem[eabus]=mbus=acc, r[0]=rbus=acc, acc=sum=abus=mor,
        addr=&ppc, eabus=xsum=addr+xbus;
    mbus.1=eabus=xsum=addr=-12, mor=mbus=r[0], r[0]=rbus=acc,
        acc=sum=bbus=acc>>2;
    bbus=mbus, abus=mor, r[1]=rbus=acc, xbus=x[1], addr=&signal,
        eabus=xsum=addr+xbus, mor=mem[eabus], sum=abus+bbus;
    mor=mbus=r[0], acc=sum=abus=sat(-mor), sign=sum.-1<0,
        _BMEM_TEMP_1:=!SIGN;
    bbus=mbus=r[1], r[0]=rbus=acc, abus=-mor, acc=sum=abus+bbus;
    abus=mor, acc=sum=sat(bbus+abus), bbus=acc>>5;
    acc=sum=abus=mor, r[0]=rbus=acc, mor=mbus=r[0], CC:=_BMEM_TEMP_1;
    sum=bbus=mbus=r[0], acc=cc?sum:acc, _BMEM_TEMP_1:=CC;
    xbus=x[1], mem[eabus]=mbus=acc, abus=mor, bbus=acc,
        sum=sat(bbus+abus), addr=&thresh, eabus=xsum=addr+xbus;
    sign=sum.-1<0, _BC:=AND(SIGN,_BMEM_TEMP,_BMEM_TEMP_1);
    ;

    BRANCH {
        _BC =>      GOTO 4;
        !_BC =>     GOTO 5;
    }

4:      /* 7 instructions */

   /*
    * pitch.k, 91: (&thresh)[j] = (&signal)[j];
    * pitch.k, 92: (&old_pp)[j] = (&pp)[j];
    * pitch.k, 93: (&pp)[j] = (&ppc)[j];
    * pitch.k, 94: (&ppc)[j] = (int ) 0;
    */

    xbus=x[1], addr=&signal, eabus=xsum=addr+xbus, mor=mem[eabus];
    xbus=x[1], mem[eabus]=mbus=mor, (void)mor, addr=&thresh,
        eabus=xsum=addr+xbus;
    xbus=x[1], addr=&pp, eabus=xsum=addr+xbus, mor=mem[eabus];
    xbus=x[1], mem[eabus]=mbus=mor, (void)mor, addr=&old_pp,
        eabus=xsum=addr+xbus;
    xbus=x[1], addr=&ppc, eabus=xsum=addr+xbus, mor=mem[eabus];
    xbus=x[1], mem[eabus]=mbus=mor, acc=sum=abus=0, (void)mor,
        addr=&pp, eabus=xsum=addr+xbus;
    xbus=x[1], mem[eabus]=mbus=acc, addr=&ppc, eabus=xsum=addr+xbus;

                GOTO 6;
```

```
5:      /* 4 instructions */

   /*
    * pitch.k, 96: (&old_pp)[j] = (&old_pp)[j];
    * pitch.k, 97: (&pp)[j] = (&pp)[j];
    */

    xbus=x[1], addr=&old_pp, eabus=xsum=addr+xbus, mor=mem[eabus];
    xbus=x[1], mem[eabus]=mbus=mor, (void)mor, addr=&old_pp,
        eabus=xsum=addr+xbus;
    xbus=x[1], addr=&pp, eabus=xsum=addr+xbus, mor=mem[eabus];
    xbus=x[1], mem[eabus]=mbus=mor, (void)mor, addr=&pp,
        eabus=xsum=addr+xbus;

                    GOTO 6;

6:      /* 26 instructions */

   /*
    *              score_2 = score;
    *              a = (&pp)[i];
    *              b = (&pp)[j];
    *              c = (&old_pp)[j];
    * pitch.k, 38: score_2 = abs(a+-b)+(int)-4<0?score_2+(int)1:score_2;
    * pitch.k, 39: score_2 = abs(a+-c)+(int)-4<0?score_2+(int)1:score_2;
    * pitch.k, 40: score_2 = abs(a+(-b+-c))+(int)-4<0?score_2+(int)1:score_2;
    *              tally_score_result = score_2;
    * pitch.k, 99: score = tally_score_result;
    *              j = j + (int ) 1;
    *              _BC = j + (int ) -6 < 0;
    */

    xbus=x[0], addr=&pp, eabus=xsum=addr+xbus, mor=mem[eabus];
    xbus=x[1], acc=sum=abus=mor, r[0]=rbus=mbus=mor, addr=&pp,
        eabus=xsum=addr+xbus, mor=mem[eabus];
    xbus=x[1], bbus=acc, r[1]=rbus=mbus=mor, acc=sum=bbus+abus,
        abus=-mor, addr=&old_pp, eabus=xsum=addr+xbus, mor=mem[eabus];
    mor=mbus=acc, acc=sum=abus=mor;
    r[1]=rbus=acc, eabus=xsum=addr=&_MEM_TEMP, mem[eabus]=mbus=r[1],
        acc=sum=abus=abs(mor);
    mbus.1=eabus=xsum=addr=-4, mor=mbus=acc;
    bbus=mbus, abus=mor, eabus=xsum=addr=&score, mor=mem[eabus],
        sum=abus+bbus;
    abus=1, bbus=mbus=mor, acc=sum=bbus+abus, sign=sum.-1<0,
        _BMEM_TEMP:=SIGN;
    acc=sum=abus=mor, r[1]=rbus=acc, mor=mbus=r[1], CC:=_BMEM_TEMP;
    sum=bbus=mbus=r[1], acc=cc?sum:acc;
```

```
    bbus=mbus=r[0], r[0]=rbus=acc, eabus=xsum=addr=&_MEM_TEMP+1,
        mem[eabus]=mbus, acc=sum=bbus+abus, abus=-mor;
    mor=mbus=acc, mbus.1=eabus=xsum=addr=-4, acc=sum=abus=-mor;
    bbus=mbus, r[1]=rbus=acc, sum=abus+bbus, abus=abs(mor);
    abus=1, bbus=mbus=r[0], acc=sum=bbus+abus, sign=sum.-1<0,
        _BMEM_TEMP:=SIGN;
    acc=sum=bbus=mbus=r[0], r[0]=rbus=acc, CC:=_BMEM_TEMP;
    sum=bbus=mbus=r[0], eabus=xsum=addr=&_MEM_TEMP, mor=mem[eabus],
        acc=cc?sum:acc;
    bbus=mbus=r[1], eabus=xsum=addr=&_MEM_TEMP+1, mor=mem[eabus],
        r[0]=rbus=acc, acc=sum=abus+bbus, abus=-mor;
    bbus=acc, abus=mor, x[2]=eabus=mbus=r[0], acc=sum=abus+bbus;
    mor=mbus=acc, mbus.1=eabus=xsum=addr=-4;
    bbus=mbus, addr=1, xbus=x[2], x[2]=eabus=xsum=xbus+addr,
        sum=abus+bbus, abus=abs(mor);
    acc=sum=bbus=mbus=r[0], mbus.1=eabus=xsum=xbus=x[2],
        sign=sum.-1<0, CC:=SIGN;
    sum=bbus=mbus, addr=1, xbus=x[1], x[1]=eabus=xsum=xbus+addr,
        acc=cc?sum:acc;
    mem[eabus]=mbus=acc, eabus=xsum=addr=&score;
    addr=-6, xbus=x[1], xsum=xbus+addr;
    xsign=xsum.-1<0, _BC:=XSIGN;
    ;

    BRANCH {
        _BC =>      GOTO 3;
        !_BC =>     GOTO 10;
    }

7:      /* 5 instructions */

   /*
    *   _BC = ! (score + - topscore < 0);
    */

    eabus=xsum=addr=&topscore, mor=mem[eabus];
    eabus=xsum=addr=&score, mor=mem[eabus], acc=sum=abus=-mor;
    bbus=acc, abus=mor, sum=abus+bbus;
    sign=sum.-1<0, _BC:=!SIGN;
    ;

    BRANCH {
        _BC =>      GOTO 8;
        !_BC =>     GOTO 9;
    }

8:      /* 4 instructions */
```

```
   /*
    * pitch.k, 106: winner = (&pp)[i];
    * pitch.k, 107: topscore = score;
    */

    xbus=x[0], addr=&pp, eabus=xsum=addr+xbus, mor=mem[eabus];
    mem[eabus]=mbus=mor, eabus=xsum=addr=&winner;
    eabus=xsum=addr=&score, mor=mem[eabus];
    mem[eabus]=mbus=mor, eabus=xsum=addr=&topscore;

                    GOTO 9;

9:      /* 4 instructions */

   /*
    *   i = i + (int ) 1;
    *   _BC = i + (int ) -6 < 0;
    */

    addr=1, xbus=x[0], x[0]=eabus=xsum=xbus+addr;
    addr=-6, xbus=x[0], xsum=xbus+addr;
    xsign=xsum.-1<0, _BC:=XSIGN;
    ;

    BRANCH {
        _BC =>      GOTO 2;
        !_BC =>     GOTO 1;
    }

10:     /* 1 instructions */

   /*
    */

    ;

    BRANCH {
        !EOS =>     GOTO 10;
        EOS =>      GOTO 7;
    }
```

Bibliography

[1] Alexander Aiken and Alexandru Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.

[2] Alexander Aiken and Alexandru Nicolau. Perfect pipelining: A new loop parallelization technique. In *2nd European Symposium on Programming*, Lecture Notes in Computer Science 300, pages 221–235. Springer-Verlag, March 1988.

[3] Jonathan Allen. Computer architecture for digital signal processing. *Proceedings of the IEEE*, 73(5):852–873, May 1985.

[4] American National Standards Institute. *American National Standard for Information Systems—Programming Language C*, X3J11–1988.

[5] Marco Annaratone, Emmanuel Arnould, Thomas Karl Richard Gross, H. T. Kung, Monica Lam, Onat Menzilcioglu, and Jon A. Webb. The Warp(SM) computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

[6] Syed Khalid Azim. *Application of Silicon Compilation Techniques to a Robot Controller Design.* PhD thesis, University of California at Berkeley, May 1988.

[7] Syed Khalid Azim, Chuen-Shen Shung, and Robert W. Brodersen. Automatic generation of a custom digital signal processor for an adaptive robot arm controller. In *The IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 2021–2024, April 1988.

[8] Alan E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120b/FPS-164 family. *Computer*, 14(9):18–27, September 1981.

[9] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, 1987.

[10] Subrata Dasgupta. Parallelism in loop-free microprograms. In B. Gilchrist, editor, *Information Processing 77*, pages 745–750, North-Holland, Amsterdam, 1977.

[11] Scott Davidson. High level microprogramming—current usage, future prospects. In *The 16th Annual Workshop on Microprogramming*, pages 193–200, 1983.

[12] Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick Wayne Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C-30(7):460–477, July 1981.

[13] D. J. DeWitt. *A Machine Independent Approach to the Production of Optimized Horizontal Microcode.* PhD thesis, University of Michigan, June 1976.

[14] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, February 1985. ACM doctoral dissertation award.

[15] Joseph A. Fisher. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources.* PhD thesis, New York University, October 1979. Available from Courant Mathematics and Computing Laboratory as DOE report COO-3077-161.

[16] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[17] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, June 1984.

[18] Joseph A. Fisher, David Landskov, and Bruce D. Shriver. Microcode compaction: Looking backward and looking forward. In *Proceedings of the National Computer Conference*, pages 95–102. AFIPS, 1981.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Fransisco, 1979.

[20] Bernard Gold and Lawrence R. Rabiner. Parallel processing techniques for estimating pitch periods of speech in the time domain. *The Journal of the Acoustical Society of America*, 46(2, part 2), 1969.

[21] R. Preston Gurd. Experience developing microcode using a high level language. In *The 16th Annual Workshop on Microprogramming*, pages 179–184, 1983.

[22] W. C. Hopkins, M. J. Horton, and C. S. Arnold. Target-independent high-level microprogramming. In *The 18th Annual Workshop on Microprogramming*, pages 137–143, 1985.

[23] Sadahiro Isoda, Yoshizumi Kobayaski, and Toru Ishida. Global compaction of horizontal microprograms based on generalized data dependency graph. *IEEE Transactions on Computers*, C-32(10):922–933, 1983.

[24] Monica Sin-Ling Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie-Mellon University, May 1987.

[25] David Landskov, Scott Davidson, Bruce D. Shriver, and Patrick Wayne Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[26] Perng-Yi Ma. The design of a firmware engineering tool: The microcode compiler. In *Proceedings of the National Computer Conference*, pages 87–93. AFIPS, 1981.

[27] Patrick Wayne Mallett. *Methods of Compacting Microprograms.* PhD thesis, University of Southwestern Louisiana at Lafayette, December 1978.

[28] Peter Marwedel. A retargetable compiler for a high-level microprogramming language. In *The 17th Annual Workshop on Microprogramming*, pages 267–274, 1984.

[29] Michael D. Poe. Heuristics for the global optimization of microprograms. In *The 13th Annual Workshop on Microprogramming*, pages 13–22, 1980.

[30] Jan Rabaey, Hugo De Man, Joos Vanhoof, Gert Goossens, and Francky Catthoor. Cathedral-II: A synthesis system for multiprocessor DSP systems. In Daniel D. Gajski, editor, *Silicon Compilation.* Addison-Wesley, 1988.

[31] Jan Rabaey, Stephen Pope, and Robert W. Brodersen. An integrated automatic layout generation system for DSP circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-4(3):285–296, July 1985.

[32] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 departmental supercomputer. *Computer*, 22(1):12–35, January 1989.

[33] B. Ramarkrishna Rau, Christopher D. Glaeser, and Raymond L. Picard. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *The 9th Annual International Symposium on Computer Architecture*, pages 131–139, 1982.

[34] Brian Richards and Markus Thaler. University of California at Berkeley. Private communication, 1989.

[35] Ken Rimey and Paul N. Hilfinger. A compiler for application-specific signal processors. In *VLSI Signal Processing, III*, pages 341–351. IEEE Press, November 1988.

[36] Ken Rimey and Paul N. Hilfinger. Lazy data routing and greedy scheduling for application-specific signal processors. In *The 21st Annual Workshop on Microprogramming and Microarchitecture*, pages 111–115, November 1988.

[37] R. J. Sheraga and J. L. Gieser. Automatic microcode generation for horizontally microprogrammed processors. In *The 14th Annual Workshop on Microprogramming*, pages 154–168, 1981.

[38] Chuen-Shen Shung. *An Integrated CAD System for Algorithm-Specific IC Design*. PhD thesis, University of California at Berkeley, May 1988.

[39] C.S. Shung, R. Jain, K. Rimey, E. Wang, M. B. Srivastava, E. Lettang, S. K. Azim, P. N. Hilfinger, J. Rabaey, and R. W. Brodersen. An integrated CAD system for algorithm-specific IC design. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, Architecture Track*, pages 82–91, 1989.

[40] Bogong Su, Shiyuan Ding, and Jinshi Xia. URPR—an extension of URCR for software pipelining. In *The 19th Annual Workshop on Microprogramming*, pages 94–103, 1986.

[41] Lars Svensson. PhD thesis, Lund University, Sweden, 1990. Expected.

[42] Lars Svensson, Mats Torkelson, Lars Thon, and Rajeev Jain. Implementation aspects of a decision feedback equalizer ASIC using an automatic layout generation system. In *The International Symposium on Circuits and Systems*, pages 585–588, Finland, June 1988.

[43] Lars Thon. University of California at Berkeley. Work in progress.

[44] M. Tokoro, E. Tamura, and T. Takizuka. Optimization of microprograms. *IEEE Transactions on Computers*, C-30(7):491–504, July 1981.

[45] Roy F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 48–57, June 1984.

[46] Steven R. Vegdahl. *Local Code Generation and Compaction in Optimizing Microcode Compilers*. PhD thesis, Carnegie-Mellon University, December 1982.

[47] Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *The 15th Annual Workshop on Microprogramming*, pages 125–133, 1982.

[48] W. G. Wood. Global optimization of microprograms through modular control constructs. In *The 12th Annual Workshop on Microprogramming*, pages 1–6, 1979.