A Compiler for Application-Specific Signal Processors

Ken Rimey and Paul N. Hilfinger
Computer Science Division
University of California
Berkeley, CA 94720

Abstract

We have built a compiler that generates code for horizontal-instruction-word, application-specific signal processors. The designer of such an application-specific processor tunes the architecture using as feedback the compiled signal-processing code. Accordingly, the compiler is user-retargetable and designed for use with a plastic, irregular style of architecture.

1 Introduction

Designers now have the option of implementing digital signal processing, control, and other dedicated systems as application-specific integrated circuits (ASIC's) to reduce unit cost. Moreover, to reduce design cost, ASIC's may incorporate programmable processors if the required sample rate is not too high. The architecture of such an application-specific *processor* can be tuned for the program that it will run.

An application-specific processor should be just fast enough for the application, as small as possible, and tunable. Reducing the processor size frees space for integrating additional peripheral circuits (or at least reduces the total chip area). Tunability is a means for both speeding up and shrinking the processor.

While programming in machine language would hinder the tuning process, so would the use of a compiler that is difficult to retarget. On the other hand, a user-retargetable compiler such as the one described in this paper can guide the process. The designer evaluates a change in the architecture by retargeting the compiler and recompiling the program to observe the effect on the instruction count.

A family of tunable processors has been developed by members of the *Lager* project[1] at Berkeley [1, 2]. These processors use a horizontal instruction word: a vector of control signals with little or no restrictive encoding. Thus they are single-level computers that execute a machine language resembling the horizontal microcode of two-level general-purpose computers. Horizontal-instruction-word processors avoid opcodes, complex instruction formats, and instruction-decoding hardware—hindrances to tuning the processor datapath.

The architectural style chosen for the datapath is a challenging one for which to generate code. While it utilizes pipelining and some parallelism, it abandons the regularity expected by compiler writers. To generate efficient code for these datapaths, we have developed the technique of *lazy data routing*, which is outlined here and detailed in a separate paper [3].

We use this technique in a compiler for a variant of C called RL. The RL compiler has been used to translate programs several hundred lines in length. It is retargeted by modifying a machine description that is read along with the source program. We have recently revised RL and the formats of the machine description and the compiler's output; this paper reflects the new versions. We are currently updating the compiler.

The RL compiler is being used to design two experimental ASIC's. Thon is developing a chip for inverse-kinematics calculations for a robot arm [4]. Svensson is developing an adaptive filter for SQAM mobile radio [5]. Both processors require architectural innovations that will test the retargetability of the compiler. In experiments with previous versions of the compiler, modifying the machine description was found to be easy. The compiler seems to be more reliable than intuition for assessing architectural changes[2].

[1] Lager also addresses high-speed applications beyond the scope of the methodology described here.

[2] For example, Thon used the compiler to evaluate parallel multipliers. Introducing a single-cycle parallel multiplier halved the static instruction count of his program, but somewhat unexpectedly, a three-cycle multiplier did almost as well.

2 Target Architectures

The target family of application-specific processors is best described by example. The *Kappa* processor architecture, designed for a robot-control application [6], has served well as a prototype for customization. The RL compiler is designed to generate code for *Kappa-like* architectures.

This section describes Kappa and, by implication, its relatives. All are horizontal-instruction-word machines and all use functionally identical control units. On the other hand, their datapaths vary in the functional units and registers provided, and broadly, in topology. Still, the datapaths share similarities on which we can base the compiler.

2.1 The Datapath

Kappa's datapath, which performs fixed-point arithmetic, is shown in Figure 1. Despite usual practice in the signal processing field, Kappa does not include a parallel multiplier. (Some of its derivatives do.) Instead, it provides circuitry for controlling serial multiplication. This circuitry is associated with the coefficient register at the bottom left of the diagram, but is not itself shown.

Because Kappa uses a fully horizontal instruction word free of restrictive encoding, the diagram generally defines how the depicted functional units can be used together: every meaningful combination of actions is allowed. There is a pipeline delay of one instruction-time associated with each register and register bank; functional units are considered to have no delay. The memory at the top left is essentially a register bank, except that one address—computed in the datapath—is used for both reading and writing.

Kappa is *irregular*, meaning that its datapath topology has been chosen to suit a particular program (the robot-arm controller) rather than to suit the compiler. The opposite extreme would be exemplified by an architecture in which all intermediate results are stored in a large, central, multiported register bank. The advantage offered by irregularity is greater latitude for tuning. The problems that it poses for retargetable compilers are ameliorated by minimizing the use of restrictive instruction encoding.

One reason for giving the compiler a pretuned architecture is to prevent it from using resources frivolously. This is the only reason, for example, behind specifying finite sizes for Kappa's register banks. Because unused registers can always be dropped after code generation, the specified sizes are, in effect, just upper bounds. Bigger simplifications—dropping the entire address arithmetic unit on the right side of the figure, for example—might be possible if Kappa were used as-is for a simpler application than the one for which it was designed.

The Cathedral-II synthesis system described by Rabaey et al. [7] relies heavily on this kind of post-compilation simplification of the datapath topology. It generates code for a completely interconnected network of functional units and drops unused busses afterwards. In contrast, in our approach the designer chooses a sparse network of busses and leaves the compiler to find multi-step routes for intermediate results. A route (leading from the functional unit that produces the result to a functional unit that uses it) may have to pass through intervening functional units. We return to the difficult task of finding routes in Section 4.2.

2.2 The Boolean and Control Units

The boolean unit and the control unit do the decision-making in the Kappa family of architectures. Their designs take advantage of the dedicated nature of the processor; both units contain 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`. 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. 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 to dynamically control, say, the loading of the accumulator (`acc`).

The control unit contains a state machine and a program counter whose outputs are concatenated to form the address for the program memory (whose address space can have gaps). A branch is performed by simultaneously clocking the state machine and resetting the program counter. With the aid of a return-address stack, the control unit provides a fully general *multi-way jump/call/return* capability.

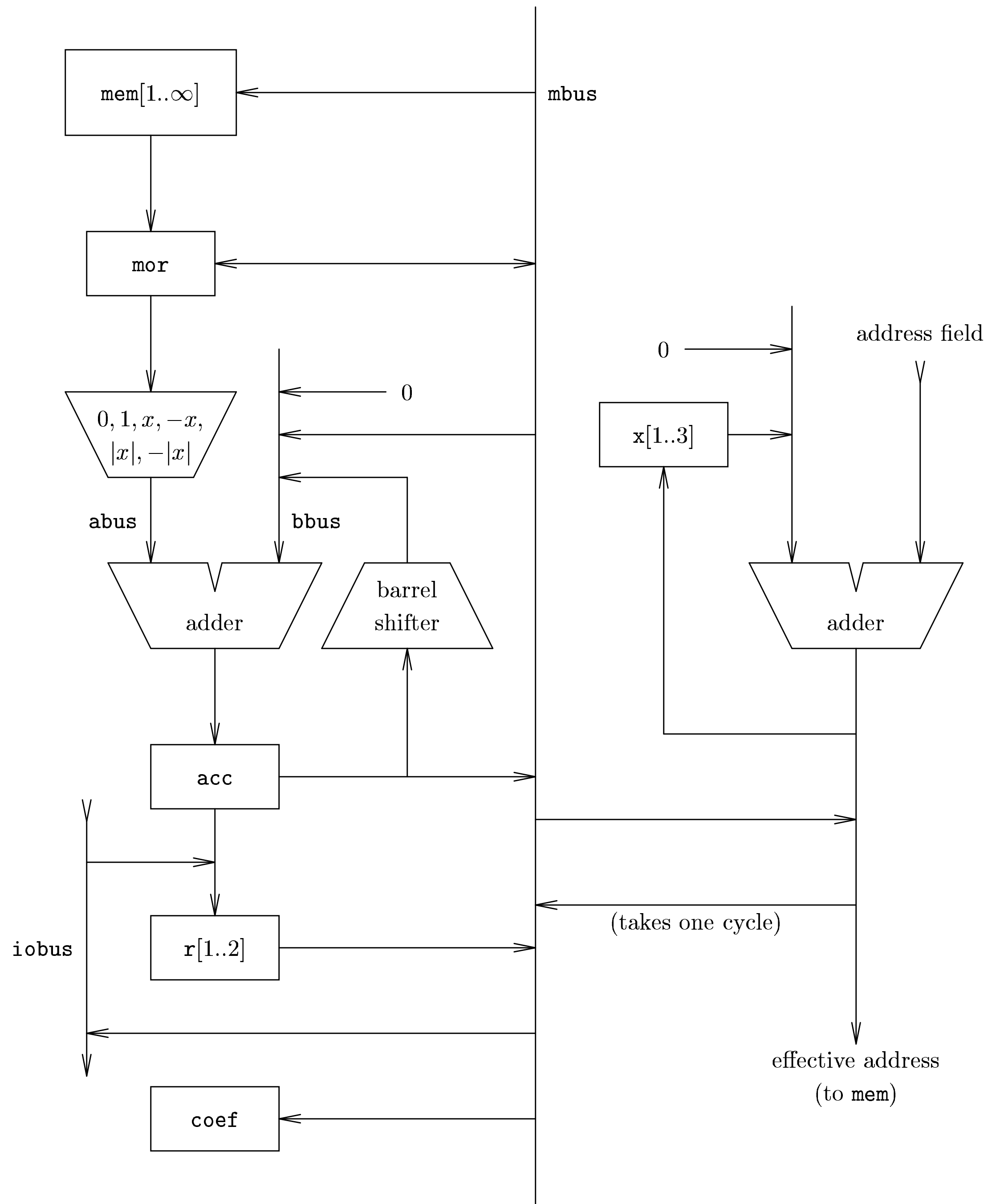

Figure 1: The Kappa datapath.

3 Using the RL Compiler

The inputs to the RL compiler are a source program and a machine description. The user provides both, but he usually does not write the machine description from scratch. The output is an assembly language program.

3.1 RL

The RL language is an approximate subset of C. To keep the compiler simple, RL includes only those features of C that correspond closely to the capabilities of Kappa and related architectures—we outlaw recursion, for example. RL includes two major extensions: fixed-point types and register type modifiers. It is therefore not strictly compatible with C. For behavioral simulation, we provide a translator that converts RL into standard C acceptable to other compilers.

Fixed-point types are a convenience for the programmer. The underlying integer arithmetic is relatively inconvenient to write by hand, partly because simple fixed-point constants correspond to huge integers, and partly because the natural multiplication for fixed-point numbers is not integer multiplication. In adding a new numerical type to a programming language, finding an elegant notation for the new constants can be difficult. In RL, all constants are typeless real numbers that take on appropriate types from context. In declarations and type casts, the fixed-point type of range $2^i > x \geq -2^i$ is denoted by `fix:`i; or if $i = 0$, just `fix`.

Register type modifiers, which generalize C register declarations, let the programmer suggest storage locations for critical variables. For example,

```
register "r" fix  y;
```

declares the variable `y` to be a fixed-point number to be stored in the register bank `r`. A reasonable default is chosen if the name of the register bank is omitted (as in Figure 2). Register type modifiers are also helpful with multiple memories, and they can be applied to pointers. For example,

```
"mem" fix  * "mem2" p;
```

declares `p` to reside in `mem2` and point into `mem`.

The sample program in Figure 2 is a trivial low-pass filter. Using the recurrence

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

it smooths the input sequence $x_1, x_2, x_3, \ldots$ to produce the output sequence $y_1, y_2, y_3, \ldots$. Instead of `main`, RL programs define the functions `init` and `loop`. This implicit form for the outermost loop will facilitate, for example, automatic insertion of memory-refresh code. The *pragma* has recently been introduced into C by the ANSI draft standard [8]. A pragma at the head of the sample program is used to specify the word width of the processor.

```
#pragma word_length 16

register fix y;

init() {
  y = 0;
}

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

/*  --Provided by the compiler.
    main() {
      init();
      for (;;) loop();
    }
*/
```

Figure 2: A simple filter described in RL.

3.2 The Register-Transfer Notation

The assembled low-pass filter is shown in Figure 3. The code consists of two straight-line segments, each of which is a sequence of instructions terminated by semicolons. Each instruction is a sequence of microoperations separated by commas. We have abbreviated "a `=` b`,` b `=` c" by "a `=` b `=` c" to make the code easier to read.

The microoperations are written in a register-transfer notation reminiscent of C or RL. Subscripting (e.g., `r[0]`) is used to refer to an element of a register (or memory) bank. Member selection (e.g., `bbus.1`) is used to refer to the value of a register or bus at a given time: *register*`.0` refers to the value of *register* in the current instruction; *register*`.1`, to the value in the next instruction; and so on.

Reasonable defaults reduce the need for the latter notation. An unqualified name means *register*`.0`, except on the left hand side of an assignment. There, it means *register*`.`*delay*, where *delay* is defined in the machine description. For example, in the microoperation

```
init:  acc=0;
       r[0]=acc;
       GOTO loop;

loop:  acc=bbus=mbus=r[0], mor=mbus, r[1]=iodata=in();
       mor=mbus=r[1], abus=mor, bbus=acc>>1, bbus.1=(abus+bbus)>>1;
       acc=bbus;
       acc=abus=mor, mor=mbus=acc;
       abus=mor, bbus=acc>>2, acc=abus+bbus;
       iodata=mbus=acc, out(iodata), r[0]=acc;
       GOTO loop;
```

Figure 3: The filter in assembly language for Kappa.

```
bbus.1 = (abus.0 + bbus.0) >> 1
```

the qualifiers on the right can be omitted, but the one on the left must remain because the *delay* attribute of `bbus` is 0. This register transfer designates a single microoperation, even though its right-hand side appears to be a compound expression. The microoperation is equivalent to "`acc = abus + bbus`", followed in the next instruction by "`bbus = acc >> 1`", except that it causes the shifter to correct for overflow in the adder.

3.3 The Machine Description

The machine description consists of declarations of registers and busses, followed by a list of the implemented microoperations.

Table 1 summarizes the information contained in the declarations of the registers and busses used in the assembled low-pass filter. Each node—bus, register, register bank, etc.—has these attributes:

- A *delay*, which was described in Section 3.2.
- A *capacity*, which is the number of values that can be stored. For anything other than a register bank, *capacity* = 1.
- A flag that identifies *static* nodes, those that retain their values from one instruction to the next. (This is unrelated to the distinction between "static" and "dynamic" registers.)

This classification system is a more general and more precise replacement for terms like *register* and *bus*. For example, although we would call both `mor` and `acc` registers, only `mor` is *static*. `acc` is really just a pipeline stage.

The list of microoperations forms the body of the machine description. Associated with each there may be information for use by other programs—the assembler, for example. Associated with exceptional microoperations there may also be compiler directives. Compiler directives impose constraints on microoperation scheduling. For example, they are used to force input and output microoperations to be scheduled in order. They are used to declare groups of microoperations to be incompatible in the rare case in which the incompatibility is not apparent from the register transfers—perhaps because it reflects instruction encoding.

Microoperations fall into two groups: transfer microoperations and function microoperations. Those that just copy values from one node to another (e.g., `acc = abus` and `mbus = r[N]`) are transfer microoperations. All others (e.g., `acc = abus + bbus`) are function microoperations. The compiler automatically combines transfer microoperations to chain together function microoperations. It automatically selects from among equivalent function microoperations when there is more than one.

The transfer microoperations define the topology of the datapath. The compiler imposes a few constraints on the designer's choice of topology, as well as on the compiler directives that may be attached to transfer microoperations.

Table 1: The busses and registers used in the assembled program.

name	delay	capacity	static?
`abus`, `bbus`, `mbus`, `iodata`	0	1	no
`acc`	1	1	no
`mor`	1	1	yes
`r`	1	2	yes

4 Implementation of the Compiler

The compiler is in two parts. The front end translates the program into successive straight-line segments of code, expressed in an intermediate language. Then, for each straight-line segment, the back end selects microoperations and packs them into instruction words. Only the back end uses the machine description.

4.1 The Front End

The front end performs a variety of routine tasks and simple optimizations, including parsing, constant folding, building the symbol table, and type analysis. Because they reflect our target machines, two additional optimizations are of note.

The first is reduction of multiplications by constants into minimal sequences of shifts, additions, and subtractions. For example, a fixed-point multiplication by 7/8 is reduced to a three-bit right-shift and a subtraction.

The second is control-flow 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. All jumps to jumps, to calls, and to returns are eliminated in the process.

For each straight-line segment, the front end passes to the back end a dataflow graph whose nodes are intermediate-language operations such as "add" and "shift-right." The front end eliminates all variables that are not live across branches by adding edges to the graph. The back end assigns the remaining variables to register banks using the register declarations from the RL program together with reasonable defaults. These variables are accessed by special "read" and "write" nodes in the dataflow graph, which itself makes no reference to particular busses or registers.

4.2 The Back End

Most of the effort in developing the RL compiler has gone into developing the algorithms used in the back end. A separate paper discusses these in depth [3]. Here, we describe the relationship of our basic approach to the approaches that have been tried elsewhere in the compiler-construction community.

The usual approach to generating horizontal code has been to separate the process into two phases. First loose sequences of microoperations are generated. Then these are packed tightly into a small 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. Good heuristics for this have been developed and thoroughly studied.

Global compaction generalizes local compaction to include movement of microoperations across branches. The best known method, *trace scheduling*, has been developed in conjunction with very-long-instruction-word (VLIW) supercomputers [9]. Unfortunately, trace scheduling is unsuitable for signal processing applications because it improves average running time only at the expense of worse-case time.

Software pipelining is a specialized technique for the compaction of loops. It rearranges the body of a loop to overlap the execution of many successive iterations. Touzeau describes the use of software pipelining in a compiler for the FPS-164 array processor [10].

Fisher, Landskov, and Shriver's paper on microcode compaction [11] is a good general introduction to these techniques.

Vegdahl critically examines this separation of code generation into two phases [12]. He concludes that some coupling of them is useful: feedback from the compaction phase is needed for effective microoperation selection. We find this issue to be particularly important in generating transfer microoperations for Kappa-like architectures. However, rather than couple the two phases, we have chosen to completely integrate them.

The back end of the RL compiler *schedules* microoperations as they are selected. This approach creates numerous opportunities for code improvement—even when it is applied just to one straight-line program segment at a time. Hence we have limited ourselves to *local scheduling*.

Our scheduler is similar to the "operation scheduler" developed by Ruttenberg and described in a paper written with Fisher and others [13]. It does *greedy* scheduling of function microoperations and *lazy* scheduling of transfer microoperations. For each node of the DAG in turn, it finds the function microoperation that can be inserted into the earliest possible instruction, and inserts it. It also finds transfer microoperations that deliver the arguments of the function microoperation, and inserts them into the appropriate instructions. This subtask is *lazy 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), we get more constraints to guide the selection. We also get a complication: the possibility that all feasible routes for the value will be unwittingly closed off by an unfortunate scheduling decision in the meantime.

We detect and avoid such disaster by maintaining a feasible route to the indefinite future, a *spill path*, for each value with as-yet-unscheduled uses. These spill paths do not represent scheduling commitments; indeed, they are continuously adjusted as scheduling proceeds. Their purpose is to identify scheduling decisions for which no accommodating adjustment exists.

Spill-path adjustment can be performed efficiently; any number of spill paths can be rerouted in a time linear in the size of the *space-time graph*. This is a directed graph whose nodes are ordered pairs $[r, t]$ such that r is a node of the datapath and $t \in \{1, 2, 3, \ldots, t_\infty\}$ is an instruction number. The quantity t_∞ is the number of instructions in the current schedule, and then some. There is an edge from $[r_1, t_1]$ to $[r_2, t_2]$ if and only if there is a transfer micro-operation that copies a value from r_1 to r_2 in $t_2 - t_1$ instruction-times. Spill paths respect the node capacities of the space-time graph; if the *capacity* attribute of the datapath node r is n, and if r is currently committed to holding k values at t, then no more than $n - k$ spill paths may pass through the space-time node $[r, t]$. By a standard construction that splits each node of the graph into two, the space-time graph with its node capacities can be converted into a space-time *network* with edge capacities. On this the spill paths form a *network flow*, to which well-known algorithms apply. In particular, if the flow can be adjusted to accommodate a unit reduction in the capacity of an edge, this can be done in a time linear in the size of the network.

5 Conclusion

We have described a class of programmable processors and a user-retargetable compiler that form the basis for a practical ASIC development strategy. The use of irregular horizontal-instruction-word architectures facilitates the tuning of the processor datapath but limits the applicability of standard code-generation techniques. To generate efficient code for diverse datapath topologies, we have developed the technique of lazy data routing.

The task ahead is to explore the architectural regime in which this technique is effective, with the aim of evaluating its applicability to future generations of application-specific processors. The experimental ASIC's already under development by Thon and Svensson include major differences from Kappa and are a first step in this direction.

Acknowledgements

This research is funded by DARPA contract number N00039-87-C-0182. We want to thank Edward Wang and Lars Thon for their comments on drafts of this paper.

References

[1] J. Rabaey, S. Pope, and R. W. Brodersen, "An integrated automatic layout generation system for DSP circuits," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 285–296, July 1985.

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

[3] K. Rimey and P. N. Hilfinger, "Lazy data routing and greedy scheduling for application-specific signal processors," in *The 21st Annual Workshop on Microprogramming and Microarchitecture*, pp. 111–115, Nov. 1988.

[4] L. Thon. Work in progress, University of California at Berkeley.

[5] L. Svensson, M. Torkelson, L. Thon, and R. Jain, "Implementation aspects of a decision feedback equalizer ASIC using an automatic layout generation system," in *The International Symposium on Circuits and Systems*, (Finland), pp. 585–588, June 1988.

[6] S. K. Azim, C.-S. Shung, and R. 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*, vol. 4, pp. 2021–2024, Apr. 1988.

[7] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, and F. Catthoor, "Cathedral-II: A synthesis system for multiprocessor DSP systems," in *Silicon Compilation* (D. D. Gajski, ed.), Addison-Wesley, 1988.

[8] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Englewood Cliffs, New Jersey: Prentice-Hall, second ed., 1988. This edition reflects the draft ANSI C standard.

[9] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, 1987.

[10] R. F. Touzeau, "A Fortran compiler for the FPS-164 scientific computer," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pp. 48–57, June 1984.

[11] J. A. Fisher, D. Landskov, and B. D. Shriver, "Microcode compaction: Looking backward and looking forward," in *Proceedings of the National Computer Conference*, pp. 95–102, AFIPS, 1981.

[12] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," in *The 15th Annual Workshop on Microprogramming*, pp. 125–133, 1982.

[13] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel processing: A smart compiler and a dumb machine," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pp. 37–47, June 1984.