Abstract
Generating high quality code for embedded processors is made difficult by irregular architectures and highly encoded parallel instructions. Rather than deal with the target machine at every stage of the compilation, a promising new methodology employs generic algorithms to optimize code for an idealized abstraction of the true target machine. This code, called reference code, is then mapped to the real instruction set by enhanced genetic algorithms. One perturbs the original schedule to find a number of alternative (parallel) instruction sequences, and the other evolves feasible register assignments, if possible, for each sequence. This paper describes the strategy for mapping idealized code into actual code. The COGEN(T) system employs this methodology to produce good code for different commercial DSPs and ASIPs.

1. Introduction

Code generation is notoriously difficult for specialized target processors with limited, heterogeneous register classes, irregular instruction sets, and substantial instruction-level parallelism. Digital-Signal Processors (DSPs), Application-Specific Instruction-Set Processors (ASIPs) and many VLIWs have these characteristics. These processors are often programmed in assembler, because available compilers cannot produce sufficiently fast or compact code, given the code size and performance requirements of typical applications. An optimizing compiler is needed to allow programmers to work at higher levels and to shorten a product’s time-to-market. During system exploration or ASIP design, it would also be very useful to have a retargetable compiler that can adapt to different prospective processors or evolving instruction sets. The code generation (“back end”) function of the compiler handles most optimizations required by specialized processors, as well as retargetability issues.

Several efforts to build complete code generators are reported in [1-5]. The compilation process is always subdivided into stages. However, decisions made at each stage impact the situations faced by later stages, often subtly, and sometimes adversely, a phenomenon known as phase coupling. If the compiler is too generic, each stage may overlook opportunities for optimization or make choices that have detrimental consequences later. The more irregular the machine, the more a generic process tends to stray from optimality. However, if the compiler employs specialized transformation algorithms to confront irregularities, retargetability is compromised. In order to produce useful code, most code generators limit their potential target processors to a specific range.

Our code generator, known as COGEN(t), [5] takes a different approach in an attempt to provide optimization over a wide range of processor architectures. Rather than focus on the actual target machine and its peculiarities from the outset, code is first produced for an idealized abstraction of the target, called a clean machine. The clean machine has the same basic capabilities and structure as the real machine, but without the troublesome restrictions on parallelism or constraints on registers. Its regular structure allows powerful, parameterizable optimization algorithms to map machine-independent intermediate code to high-quality, maximally parallel code for the clean machine. This “optimized” code for the clean machine is known as reference code. COGEN(t) uses mostly original algorithms to produce reference code. However, in principle other algorithms, or even another compiler, could be employed instead. The methodology of interest here is about getting from reference code to the real (encoded and irregular) instruction set.

The point is that COGEN(t) defers consideration of instruction set irregularities until (almost) the final stage of code generation. Only then does a novel three-phase process, called Shake And Bake (SAB), confront detailed machine constraints by mapping optimized reference code from the clean machine to the actual instruction set. And it does so in the same way for every applicable processor architecture! This standard mapping process, combined with generic optimization algorithms to produce reference
code, provides retargetability without sacrificing optimization. This paper is about this mapping process.

Shake and Bake is useful for instruction sets with instruction-level parallelism, no matter how highly encoded. In general terms, mapping to valid instructions of the target machine is done by generating a number of minor perturbations of the reference code and measuring how closely these variations meet the constraints of the target instruction set. The closest match is chosen, and any required fixes are made. If the target instruction set closely resembles the clean machine, the mapping is immediate and the code quality excellent. As the machine becomes less regular, the perturbations may become more drastic, and more repair code may be introduced. The three mapping modules, called SHAKE, AND, and BAKE deal, respectively, with the issues of (re)scheduling, instruction selection, and register assignment.

SHAKE employs an Enhanced Genetic Algorithm (EGA, described later) [6-8] to “stretch” and perturb separate parallel operation streams in an existing “correct” (parallel) schedule. This causes different combinations of operations to align on the same scheduling step, so that different combinations of (concurrently scheduled) operations become candidates for co-residing in the same (parallel) instructions. In highly encoded instruction sets, the number and choice of primitive operations collected into a parallel instruction may have a profound impact on allowable registers. AND verifies that the suggested parallel instructions are not always orthogonal and may further complicate code generation. The constituents of the target machine are typical parallel DSP instructions. Instruction 3 comprises the loop body and multiplies the contents of registers X0 and Y0, adds the resulting product to accumulator A, retrieves new operands from data memories X and Y into registers X0 and Y0, respectively, and updates address registers R0 and R4 using auto-increment for each. The loop body contains operations from two iterations folded into a one parallel instruction. Standard optimizations, like loop unrolling and retiming happen long before Shake and Bake.

(a) double FIR_filter(in double A[], in double B[], in int tap) {
   int l;
   double sum=0;
   for(i=0;i<tap;i++)
      sum += A[i] * B[i];
   return sum;
}

(b) ALU X-Memory Y-Memory
   CLR A X:(R0)+, X0 Y:(R4)+, Y0 [1]
   REP #N-1 [2]
   MAC X0, Y0, A X:(R0)+, X0 Y:(R4)+, Y0 [3]
   MACR X0, Y0, A [4]

Figure 1: Partial C and M56001 assembler code for N-Tap FIR Filter.

Instruction-level parallelism makes generating efficient code difficult. A number of register classes, each having specific roles and a small number of registers, further complicate code generation. The constituents of parallel instructions are not always orthogonal and may interact to preclude or require specific register assignments. The instructions may be so highly encoded that only certain combinations of registers may be used, and allowable combinations may depend on the particular operations and/or the amount of data movement being done in parallel.

For example, consider the ST950 [10] with its highly encoded 16-bit instruction words. It has four major data-handling register classes: one product register to hold the result of multiplication; two accumulators, which hold the results of other arithmetic operations; and four operand registers, which can hold data fetched from memory. Operand registers are classified as left or right, depending on their connection to ALU input ports. The multiply instruction comes in five flavors with five different opcodes, depending on the amount of parallelism allowed. Figure 2 shows four of the five versions. The multiply instruction has its operands classified as “left” or “right”.

Section 2 explores the code generation problem in greater depth. Section 3 contains a formal description of the clean machine model, reference code, and the overall COGEN(t) methodology. Section 4 explains the specific functions of Shake, And and Bake. Section 5 describes the overall mapping process, and results are presented in Section 6.

2. The Problem of Code Generation

DSP architectures frequently have dual data memories, independent address calculation units, one or more register-register arithmetic units, and instruction-level parallelism (ILP). A single instruction can concurrently load registers from one or two memories, perform an arithmetic operation on current register values, and advance one or two address registers through their respective data streams. DSP processors are also optimized for multiply-accumulate and shift operations, which they can perform in the same complex instruction. Fig. 1(a) shows C code for implementing an N-Tap FIR filter. Figure 1(b) shows a portion of the resulting assembler code for the M56001[9]. Instructions 1 and 3 are typical parallel DSP instructions. Instruction 3 comprises the loop body and multiplies the contents of registers X0 and Y0, adds the resulting product to accumulator A, retrieves new operands from data memories X and Y into registers X0 and Y0, respectively, and updates address registers R0 and R4 using auto-increment for each. The loop body contains operations from two iterations folded into a one parallel instruction. Standard optimizations, like loop unrolling and retiming happen long before Shake and Bake.
because of register asymmetry. Its most flexible form has no parallel operations and allows any register to supply the left input. However, the right input may come from only a right operand register or an accumulator. The output must enter the product register. The most restrictive form of multiply provides maximum parallelism: two concurrent loads from its two data memories. However, besides the product register, this form permits only operand registers suitably segregated into "left" and "right". Other multiply variants fall between these extremes of parallelism and register flexibility. Some disallow specific register combinations. This machine never permits concurrent stores, although two loads or a load and store can be done concurrently.

The Motorola 56001 has a longer instruction word with more orthogonal instruction components. It corresponds more closely to its clean machine model and enjoys almost immediate success during the final mapping process. Nevertheless, it has several small register classes and some of its own restrictions. One example is the inability of the multiply instruction to use an accumulator as an input operand, although its output always enters an accumulator.

3. Evolution of code in COGEN(t)

3.1 The Clean Machine Model

Basic operations are mapped to domains. A domain corresponds to an operational portion of an instruction and represents a separate resource, such as an ALU, address unit or particular memory. Domains model the potential parallelism of an instruction set. Their exact meaning depends on how the instruction set is structured. For example, the Motorola 56001 (Fig. 1) has an arithmetic domain (MACC), a domain for each of its memories (X and Y), plus domains for its two addressing streams. A sixth domain could be used for control operations (REP).
Other machines may include domains for a conditional execution mask, floating point operations, branches, or operation modes (like trunc or round). Register and constant fields only supply values to operations and do not themselves constitute domains. Consider Fig. 2, where four domains are evident: the multiply unit, the adder (ALU), and the left and right memories. A VLIW may have a number of domains with equivalent functionality (say several ALUs); such a set constitutes a **domain type**.

An **atomic operation** is a fundamental operation, like “add” or “load”. Each atomic operation associated with a domain type defines a **basic operation**. These are the operations encountered by Shake and Bake. For example, a VLIW might include two arithmetic adds and an address register update involving addition. All three are atomic *add* operations; the two arithmetic *adds* are instances of one basic *add*-type operation, and the register update is a different basic *add* op.

Each domain in these examples implies a distinct type of functionality (and possibly distinct register classes associated with it: address, accumulator, operand, or state registers). If multiple domains provide overlapping functionality, the same basic operation may initially be a candidate for more than one domain, but will eventually be assigned to a specific domain, through binding to a particular unit. Memory accesses of spilled values provide an example when the machine has two data memories. The relevant memory domain is uncertain, until after the value is bound to one of the memories. Likewise, operations are bound to one of several similar functional units.

\( B = \{b_i\} \) are the **basic operations** implemented by the machine. Each domain \( d_j = \{b_i\} \cup \{b_0\} \) is a particular set of basic operations that always includes \( b_0 \), the no-op (*nil*). The same operation may appear in more than one domain. The **clean machine instruction set** is simply the Cartesian product of all domains: \( C = d_1 \times d_2 \times \ldots \times d_p \).

An **idealized instruction**, \( m_i \in C \), is a vector of basic operations where each is from a different domain: \( m_i = \{b_1, b_2, \ldots, b_p\} \), where \( b_i \in d_i \). Note that \( C \) has only one restriction on parallelism:

*an instruction can contain at most one basic operation for each domain*  \hspace{1cm} (1)

The **actual machine instruction set**, \( M \), is a subset of \( C \) in which certain operation combinations are not allowed. Thus \( M \subseteq C \). In a highly encoded instruction set, many domain or operation combinations may be forbidden. Nevertheless, all domains are still “present”, even if not represented by an action within the instruction. (“*Nil*” is the default value for every domain. It fills unallocated positions in instructions and may not always be explicitly shown.)

Consider the four “multiply” instructions in Fig. 2. Assume the corresponding instruction set has four domains: left(X) memory; add/subtract; multiply; right(Y) memory. These instructions appear as follows in the clean machine notation:

*\( m_1 = \{\text{nil, nil, Mpy, nil}\} \)**
*\( m_2 = \{\text{nil, nil, Mpy, Load}\} \)**
*\( m_3 = \{\text{nil, Add, Mpy, Store}\} \)**
*\( m_4 = \{\text{Load, nil, Mpy, Load}\} \)**

The instruction \([\text{store, nil, nil, store}]\) is legal in the clean machine, but not in this particular \( M \).

A **block of clean machine code** is a single sequence of instructions from the clean machine instruction set. It typically models a basic block or several consecutive basic blocks that comprise a trace with multiple entry and/or exit points. A trace is a sequence of basic blocks with fairly high probability of flow through. Imagine a rectangle whose interior is a grid (Fig. 3a.) Each row is one step in the code block. The columns represent all domains that apply to the machine. The entries in each cell are \( b_i \in B \). Entries in row \( n \) comprise instruction \( m_n \). Fig. 3b shows an abstract instruction, and Fig. 3c shows a real instruction with typical basic operations.

\[_\text{Figure 3: Visualizing the clean machine.}\]
When clean machine code is built, related operations convey their values in registers of a certain type. Registers having similar connectivity and function comprise a register class. Clean machine register classes correspond to register classes of the actual machine in their type, number and relationship to basic operations. However, all registers in a class are considered equivalent, interchangeable, and unconstrained by the context in which they are used – which may not be the case in reality. Specifically, let $R_1, R_2, \ldots, R_m$ be register classes of the target and clean machines. A register may belong to more than one class. Every basic operation has certain register sets that can be chosen for its inputs and outputs, each of which is a union of selected register classes. For example, operation $b_i$ may be binary, with inputs labeled $b_{i1}$ and $b_{i2}$, and output $b_{i3}$. Define $b_{ij}$ to mean port $j$ of $b_i$. The corresponding valid register sets (in both the clean and actual machines) are $S_{ij}$ and $S_{ji}$, and output $S_{i0}$, where $S_{ij} = \bigcup_{K} R_k$ for some index set $K$. However, when operations are combined in parallel instructions, the real machine may impose complex restrictions on which registers can be used for which ports of constituent basic operations. The clean machine does not! It treats domains as orthogonal and retains the original allowable register sets, $S_{ij}$, regardless of parallelization.

### 3.2 From the DFG to Reference Code in COGEN(t): a fast tour

In COGEN(t) the parser generates a control structure that connects basic blocks. Each basic block has its own data flow graph (DFG) that contains machine-independent atomic operations, like “add” and “shift” for source language operations, or “load” and “store” for memory access. Edges in each DFG denote data dependencies (and required orderings). Atomic operations having instruction support map directly to basic operations of the target machine. Two types of operation replacement support retargetability: First, any atomic operation that is not implemented by a basic machine operation is replaced by an equivalent combination of supported basic operations. Then certain configurations of atomic operations may be combined into supported complex operations via pattern matching, “multiply-and-accumulate” being a familiar example. All operations at this point are basic operations of the target machine; all are members of $B$. However, they are not yet combined into the parallel instructions of the target machine.

Next, basic blocks are partitioned into traces. Within non-trivial traces, ordering constraints are inserted to prevent inappropriate movement of code between basic blocks. Scheduling and binding occur on an entire code block (trace) to place operations in suitable domains and on steps that respect all ordering constraints. This means assigning all operations to distinct cells on the “grid” in appropriate ways. Define $t(b_i) = s$ to be the (integer) step on which operation $b_i$ is scheduled. Besides constraint (1), the scheduling/binding algorithm enforces:

\[ b_i \text{ must be bound to a domain } d_j \text{ for which } b_i \in d_j \quad (2) \]

\[ \text{if } b_i \rightarrow b_j \text{ (ordering), then } t(b_i) < t(b_j) \quad (3) \]

The scheduling method in COGEN(t) employs an EGA to generate a pool of (10-40) feasible schedules, meeting all constraints. Arithmetic operations are scheduled. Each candidate schedule is evaluated on a number of criteria in an attempt to balance competing considerations. These include schedule length, locality in the use of alternative units, register demand, compatibility with neighboring traces, and order of access to array values (for systematic addressing). Criteria are weighted based on availability of resources on the machine, i.e., “tuned” for retargetability. The best schedule is pursued in later stages, but its closest competitors are retained as alternates. Some of these may represent an unrolled or folded loop or alternative bindings to equivalent units in a VLIW.

Arrays are assumed to be memory-resident; their associated load and store operations are scheduled in ways that promote systematic traversal through the arrays. Scalar variables are assumed to be register-resident, until register pressure causes spills to memory. Spills introduce additional store and load operations, plus ordering constraints to keep the store and load ops properly related. Another EGA assigns arrays and spilled values to one of two memories. Then for each data memory, the scheduled strands of related load and store operations are merged into a complete access schedule for that memory. At this point, address register classes are associated with memory access tasks, and code is introduced to manipulate their values, hopefully using supported address modes. Although register demand in each register class determines the need for spills and address functions, final register assignment must await BAKE. The clean machine treats each register class as wholly and freely available, which the target machine may not.

The entire process schedules and binds operations to a few domains at a time: first arithmetic domains are populated and scheduled; then memory (and immediate constant) related domains are scheduled (with loads and stores); finally addressing operations are determined. When a code block is complete, boundary code is added to interface with neighboring traces. At each stage after the arithmetic core is completed, operations are introduced only as needed. Occupants of previously “optimized” domains may have to be moved to make room for newly added loads, stores and addressing operations. Operations never leave the domain to which they were originally bound; interrelated clusters of operations move “vertically” together. Thus, all previously enforced ordering constraints are retained as new operations are inserted. At each stage of code generation, the current
reference is highly “compact”; each successive stage may “decompact” existing code to maintain constraints. This trend continues in SHAKE.

An important feature of our compilation methodology is the use of a stochastic process (usually an EGA) to quickly find a feasible solution to the subproblem at each stage of compilation. Multiple applications of the EGA yield a pool of feasible but different solutions, all of which are quickly evaluated by a number of simple criteria. The selection criteria are weighted differently for different architectures, and more than one good candidate may continue to the next stage. We maintain many, quickly derived alternative solutions, rather than labor over one trajectory toward optimization. This emphasis on “breadth” rather than solution “depth” accommodates a wide variety of instruction sets and provides resilience to phase coupling and other anomalies. Again, this strategy is central to Shake and Bake, although the reference code could have been derived in more conventional ways.

4. From Reference Code to Real Code: the task of Shake and Bake

4.1 SHAKE: rescheduling to enable mapping

When SAB begins, the reference code is compact and correct in terms of the clean machine. Enough spills have been introduced to ensure feasible use of (idealized) register classes. All operations within each domain are permanently bound to the domain and permanently ordered within the domain. However, scheduling on specific steps is provisional. SHAKE performs localized rescheduling within domains, while preserving all ordering constraints among operations. It uses an EGA to generate a "vertical" redistribution (a “shake”) of operations. This causes different basic operations from different domains to fall on the same row of the grid. Operation alignment refers to such a “lateral” grouping of basic operations. It is an artifact of scheduling: if \( t(b_j) = t(b_i) \), then \( b_i \) and \( b_j \) happen to be “aligned” and therefore in the same candidate instruction. SHAKE vertically “stretches” and “compresses” sections of clean machine code probabilistically, hoping to produce candidate instructions that are all in M. SHAKE is applied repeatedly, producing several blocks of “correct” clean machine code that differ primarily in their internal operation alignments. All observe constraints (1), (2) and (3). In order to produce a useful variety of alternative alignments, SHAKE produces some schedules having the original (minimal) length, plus some having 1 or 2 additional steps. The extra leeway allows more vertical movement and more varied alignments. SHAKE decompacts some instances of the highly parallel reference code, since clean machine instructions containing less parallelism have a better chance of belonging to the target machine.

4.2 AND: instruction selection and register considerations

The task of AND is to check the feasibility of each block of clean machine code against the allowable alignments of the machine’s instruction set. If all alignments are in M, the current clean machine code block may be accepted as a candidate for later exploration. Otherwise that code block is discarded. Among surviving candidates, the detailed schedules are frozen on specific “steps”, as are the legal machine instructions. Thus for each block, AND guarantees:

\[
\forall n, m_n \in M \quad \text{(allowing us to say “code block”)} \quad (4)
\]

Each edge in the DFG has an associated register set, from which its final register must be chosen. These register sets may be progressively restricted as optimization algorithms assign specific registers to particular uses and eliminate certain register combinations. AND deals with aspects of the restricted-register problem. In particular, for each basic operation, \( b_i \), within a valid machine instruction, port \( b_{ij} \) of that operation may require its register to belong to a restricted register set: \( S_{ij} \subseteq S_{ij} \). If \( b_i \) produces a value (at output port \( j \)) required as input by \( b_k \) (at port \( k \)), the restricted register set available for the edge from \( b_{ij} \) to \( b_{hk} \) will be \( S_{ij} = S_{hk} = S_{ij} \cap S_{hk} \). The superscripts, 1 and 2, indicate whether the restriction to \( S \) is based on one or more instructions.

AND computes \( S_{ij} \) for every edge that carries a data value. If \( S_{ij} = \emptyset \), called register mismatch, AND inserts a copy operation on that edge, whose incoming register set is \( S_{ij} \) and whose outgoing register set is \( S_{hk} \). This ensures the possibility of conveying every data value through some (sequence of) allowable register(s). New copy operations may extend the schedule; if their number is too high, the code block is rejected as a candidate. Thus, AND guarantees:

\[
\text{If } \exists \text{ edge from } b_{ij} \text{ to } b_{hk}, \quad S_{ij} = S_{hk} \neq \emptyset \quad (5)
\]

For values having multiple destinations, the input register sets of all destinations are intersected to determine the necessity of introducing a copy operation. This ensures that the unique output from \( b_{ij} \) can be placed in a single register initially. In Fig. 2, if the output from the load (P4) of template 2 is used as an input to P4 of template 3, \( S_{ij} = S_{34} = \{R0,R1\} \). But if the same value is needed at port P5 of template 3, there is a null intersection, a register mismatch.
Figure 4 shows an example (using a commercial DSP) of how SHAKE may have to extend the schedule and reduce parallelism to find a feasible configuration. Figure 4(a) shows a schedule that has two register mismatches. Since there is no room for realignment within four steps, SHAKE increases the schedule length by one step. The 5-step alignments shown in Fig. 4(b) and 4(c) each has a single (but different) register mismatch. In Fig. 4(c), AND tries (unsuccessfully) to resolve the register mismatch by including a copy on step 2. Expansion to six steps (Fig. 4(d)) eliminates all mismatch problems.

4.3 BAKE: final register assignment

Within each code block, the instructions are valid, and potential registers are available for every connecting edge. However, some other constraints are still outstanding. At the (successful) conclusion of BAKE, each edge will have a specific register assigned, without producing any register conflicts or violating any remaining constraints. However, finding a complete feasible assignment is the most difficult part of mapping to the target machine. The register constraints surrounding a tightly encoded parallel instruction can be so severe that only a few combinations of registers are legal. The registers assigned at each instruction apply to all edges leading to and from that instruction and, as a consequence, affect choices available for many other instructions. All instructions are interrelated in subtle ways, and finding a register assignment that applies throughout a block may be an elusive task – or even impossible.

BAKE employs an EGA to generate populations of candidate register assignments for an entire block of instructions. The genetic algorithm evolves better and better assignments in an attempt to find one that satisfies all the constraints. This process is repeated for each block of code until a satisfactory register assignment is found or until all candidate blocks have been tried. Define \( r_{ij} \) to be the register assigned to (input or output) port \( j \) of basic operation \( i \). The most basic constraint is that some register must be assigned

\[
\forall \text{ ports, } j, \text{ of basic operation } i: \text{ some register } \quad r_{ij} \in S_{ij} \text{ must be assigned} \quad (6)
\]

AND has already dealt with how membership in the same parallel instruction may restrict register sets of individual basic operations; it may have rejected some candidates or inserted \textit{copy} operations into others. In highly encoded instruction sets, external constraints may further limit register choices within a single instruction. External constraints cause register choices for one port to restrict register choices at other ports, even among \textit{different} basic operations – simply because they happen to
align in the schedule. Figure 2 contains several examples. Case 1 explicitly forbids certain register combinations. Cases 2 and 3 require the same register at two ports. Case 3 has two constraints in which the register at one port dictates the register at another. In general, relationships between two ports (of an entire instruction) can be expressed as one or more implications of the form:

\[
\text{Within } m \in M: \text{ if port } b_{ij} \text{ is assigned } r_{ij} \in S^*_{ij} \subset S^2_{ij}, \text{ then port } b_{ef} \text{ must be assigned } r_{ef} \in S^*_{ef} \subset S^2_{ef} \quad (7)
\]

Here \( S^*_{ij} \) is a proper subset (possibly one register) of the (already restricted) register set allowed at port \( b_{ij} \), and similarly for \( b_{ef} \). Such constraints further restrict constraint (6). More complex cases exist, requiring more elaborate implications, but these are rare.

The remaining constraints are more familiar and apply to all instruction sets. They are, respectively, that: (8) the same register must be assigned throughout each edge connecting basic operations; (9) exactly one register must be assigned to each output port; and (10) no register can be assigned simultaneously to edges from different (conflicting) sources.

\[
\text{If } \exists \text{ edge from } b_{ij} \text{ to } b_{hk}, r_{ij} = r_{hk} \in S^2_{ij} \cap S^2_{hk} \quad (8)
\]

\[
\text{If } \exists \text{ edges from } b_{ij} \text{ to } b_{hk} \text{ and from } b_{ij} \text{ to } b_{ef}, \\
\text{ then } r_{ij} = r_{hk} = r_{ef} \quad (9)
\]

\[
\text{If (} \exists \text{ edge from } b_{ij} \text{ to } b_{hk} \text{ and} \\
\text{ (} t(b_i) \leq t(b_j) \text{ and } (t(b_j) < t(b_h) ) \text{),} \\
\text{ then } \forall \text{ output ports, } f: r_{ij} \neq r_{ef} \quad (10)
\]

Constraint (5), enforced by AND, ensures a non-empty set intersection in constraint (8). However, constraint (7) may cause these sets to be further restricted; an empty intersection here is grounds for rejecting the code block. Constraint (10) avoids register conflict, in which one value would be (erroneously) placed in a register before that register’s previous value had reached all its intended destinations. It says that if \( b_e \) lies between \( b_i \) and some operation in the schedule that is awaiting the value from \( b_{ij} \), \( b_e \) cannot alter the register updated by \( b_{ij} \).

5. A Look Inside Shake and Bake

5.1 Constraint Satisfaction

The entire process is shown in Fig. 5. SHAKE is applied several times to the original clean machine code block. It produces some alignments with minimum length schedules and others that contain a bit of scheduling leeway. AND evaluates each resulting code block. Those containing invalid alignments and those with too many edge mismatches are rejected. Remaining edge mismatches are repaired by inserting copy operations. BAKE then tries to find a totally feasible register assignment in each surviving code block. As soon as BAKE finds one, it stops. If none of the candidate code blocks has a feasible assignment, a final repair may include both spill and copy operations on the problem edges. The result is resubmitted to SAB for verification, but no further repairs are permitted. Alternatively, SAB may be restarted – with either more leeway in the schedule length or a different initial clean machine code block.

By enforcing their respective constraints, each of our three modules evolves clean machine code into code for a fully encoded target machine. The task of SHAKE is to find a set of values for \( t(b) \) that satisfy constraints (1) through (3). (Since it never revises prior bindings of operations to domains, constraint (2) is satisfied a priori.) In addition, SHAKE can consider special distance and

![Figure 5: Control flow through SAB.](image-url)
timing constraints between operations in the schedule; they take the form:

$$t(b_i) - t(b_j) \ rln \ constant, \ where$$

$$rln \in \{=, <, \leq, >, \geq, \neq\} \quad (11)$$

Operations can be kept within maximum schedule length, \(T\), by including: \(T - t(b_i) \geq 0\). AND verifies constraints (4) and (5), and it may insert copy operations to potentially satisfy (5). The task of BAKE is to find a set of values for \(r_j\) that satisfy constraints (6) through (10).

The problems of SHAKE and of BAKE are formulated as Constraint-Satisfaction Problems (CSP); AND is mostly a constraint verifier. In a pure CSP the problem is to find an instantiation of variables with finite domains, such that all constraints prescribed for the variables hold. CSPs, however, are NP-complete[11], and although heuristics have been found useful in solving them, complete search algorithms are subject to combinatorial explosion. However, the enhancements to the traditional GA enable our EGA to solve pure CSPs much faster than its familiar cousin.

Constraint satisfaction may not seem like optimization. However, we begin with "optimized" code and are trying to find an equally tight schedule that satisfies the additional constraints of the real machine. The final result will never be shorter than the reference code. All we can hope is to find a feasible arrangement without having to lengthen the code too much. That is why the initial schedule is given some small (1 or 2-step) margin for growth.

5.2 The Enhanced Genetic Algorithm

A classical genetic algorithm evolves a population of candidate solutions by combining current candidates into new ones (through crossover) that may combine the benefits of its respective "parents". Through successive generations, older and less suitable solution candidates are removed, and the average suitability (fitness) of the population tends to improve. In order to maintain diversity in this stochastic parallel search process, occasional "mutations" randomly alter a member of the population. The EGA is targeted to solving CSPs efficiently. Each "chromosome" is a string of integers that represent values of the variables in the constraint specification. Each individual value is kept within the range allowed for its respective variable. The "fitness" of a candidate string is measured by the number of constraints it satisfies. The EGA promotes the mutation operator into a primary tool for adjusting those variables that contribute to unsatisfied constraints. Directed mutation is applied aggressively, but only to those variables that appear most often in violated constraints. While this promotes rapid convergence to feasible regions of the search space, another new strategy maintains diversity in the population. This technique, called partial elitism, first applies crossover, then chooses the two most fit from among the two parents and two offspring, and then applies directed mutation to the two strings selected. Discovery of a candidate solution that satisfies all constraints terminates the process. These techniques combine to offer very rapid solutions to CSPs without requiring any special parameters or tuning for different problems. (Discussion of the entire process would required too much space here. Refer to [6-8] for more information relating to EGAs.) Generating a pool of feasible solutions is simply a matter of running the EGA multiple times on the same problem. The stochastic nature of the EGA results in different solutions being generated whenever more than one solution is possible, thus providing a number of real alternatives in a short time.

6. Results

COGEN(T) is a prototype that is still under development. Nevertheless, preliminary results indicate that the methodology is capable of evolving high quality code for some difficult, highly encoded instruction sets. The mapping methodology described here has successfully mapped the same clean machine reference code to optimal assembly code for two different DSPs: the reasonably regular M56001 and the very non-orthogonal and constrained ST950.

Table 1 shows how SAB performs on the Biquad filter. In order to challenge our algorithms, all constants were fetched from memory (vs. immediate), which produced a large number of load operations. The first half of Table 1 shows how SAB handled the clean machine (11 step) reference code for each machine without coalescing any original operations using patterns. The bottom half shows the result when pattern matching combined multiply and add operations to produce 8-step reference code. (The point is simply to show the mapping at work, using different DFGs.) Here SHAKE produces pools of thirty alternative code blocks. The next two columns show how many of these conform entirely to valid instruction sequences and how many copy operations were inserted per block to make register assignment possible for each edge. BAKE was able to find feasible register assignments in all cases for the M56001, and in half the cases for the ST950 instruction set. The range of final schedule lengths comes from schedule extension by SHAKE and insertion of copy operations by AND.

Table 2 indicates COGEN(t)'s ability to handle a wider variety of different architectures. Configurations 1 through 8 reflect ASIPS with different numbers of registers and different capabilities, resulting from the ways architecture components are interconnected. Many are based on examples in Leupers [1]. Three test cases
(T1, T2, and T3) were used. Note that the two real architectures had copy operations inserted by AND to compensate for restrictions that the other machines did not exhibit. On many machines, like the ST950, register-to-register copies cannot be performed in parallel with other operations.

SHAKE employs two strategies to deal with bizarre constraints. The first strategy is to realign the operations to form different parallel instructions. The second is to extend (decompact) the schedule to decrease instruction-level parallelism -- which increases the number of "original" instructions. Decreasing parallelism tends to reduce the need for additional copies. However, the additional number of "useful" instructions may rival the number of copy instructions avoided. Table 3 shows an example in which SAB is used to map reference code for a typical signal processing algorithm to the ST950. As the allowed schedule length increases, the number of required copies decreases. Here the total number of steps is the sum of the original schedule length and the additional copies. The large number of copies (12) required for the initial, maximally parallel, 17-step schedule is a result of the highly encoded instruction set of the target machine. As the schedule is extended to 18 and 19 steps, only 7 and 6 copies, respectively, are required. These two schedules represent the best overall (25-step) schedules. Beyond 20 steps, the number of copies saved does not compensate for loss of parallelism.

<table>
<thead>
<tr>
<th>Patterns?</th>
<th>DSP</th>
<th>SHAKE Alignments</th>
<th>AND Feasible Copies</th>
<th>BAKE Feasible</th>
<th>Steps in schedule Best</th>
<th>Steps in schedule Worst</th>
<th>Time per schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>M56001</td>
<td>30</td>
<td>29</td>
<td>29</td>
<td>11</td>
<td>13</td>
<td>3.0s</td>
</tr>
<tr>
<td>yes</td>
<td>ST950</td>
<td>30</td>
<td>21</td>
<td>15</td>
<td>14</td>
<td>18</td>
<td>3.1s</td>
</tr>
<tr>
<td>no</td>
<td>M56001</td>
<td>30</td>
<td>30</td>
<td>29</td>
<td>8</td>
<td>11</td>
<td>2.0s</td>
</tr>
<tr>
<td>yes</td>
<td>ST950</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>11</td>
<td>13</td>
<td>4.3s</td>
</tr>
</tbody>
</table>

Table 1: Typical results generated by SAB at each step of the mapping process.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>M56001</th>
<th>ST950</th>
<th>ASIP1</th>
<th>ASIP2</th>
<th>ASIP3</th>
<th>ASIP4</th>
<th>ASIP5</th>
<th>ASIP6</th>
<th>ASIP7</th>
<th>ASIP8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 step</td>
<td>11 step</td>
<td>7 step</td>
<td>7 step</td>
<td>7 step</td>
<td>8 step</td>
<td>7 step</td>
<td>7 step</td>
<td>7 step</td>
<td>8 step</td>
<td>8 step</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 step</td>
<td>9 step</td>
<td>6 step</td>
<td>6 step</td>
<td>6 step</td>
<td>6 step</td>
<td>6 step</td>
<td>6 step</td>
<td>6 step</td>
<td>7 step</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 step</td>
<td>27 step</td>
<td>17 step</td>
<td>16 step</td>
<td>16 step</td>
<td>20 step</td>
<td>16 step</td>
<td>16 step</td>
<td>18 step</td>
<td>18 step</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Final schedule lengths when mapping code to DSPs and ASIPs.

<table>
<thead>
<tr>
<th>ALLOWED SCHEDULE LENGTH</th>
<th>REQUIRED COPIES</th>
<th>TOTAL NUMBER OF STEPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>12</td>
<td>29</td>
</tr>
<tr>
<td>18</td>
<td>7</td>
<td>25</td>
</tr>
<tr>
<td>19</td>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>26</td>
</tr>
<tr>
<td>23</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 3: Schedule length versus register mismatches.
7. Conclusion

In this paper we have presented a novel strategy for providing both optimization and retargetability with the same code generator. It is based on first producing high quality reference code for an unconstrained, orthogonal model of the (parallel) target machine. This is followed by mapping to the fully-constrained, real target machine via Shake and Bake. The mapping adapts the reference code in minor, stochastically determined, ways so that little or none of its quality is lost, even when the target instruction set is highly encoded and parallel.

The focus is the final mapping -- not optimization. The optimization has already been done before Shake and Bake begins. The problem here is to determine a sufficiently compact set of parallel instructions that still permits assignment of registers to all edges connecting operations while contending with all the restrictions that result from instruction-level parallelism, restricted register sets and encoding anomalies. In particular, alternative operation alignments are generated until valid instructions emerge and register assignments are tried until one is found that meets all constraints of the target machine -- no matter how restrictive. The prior optimization (of instruction numbers, say) will not be affected, unless no mapping can be found having so few instructions. When necessary, previous optimization is relaxed in order to discover a mapping, but only by an amount controlled by parameters (1 or 2 steps usually).

The advantage of performing optimization on an idealized machine is that faster and more generic algorithms can be used -- possibly on a wide variety of architectures. Shake and Bake compromises the optimization achieved by earlier stages only if the clean machine model is too different from the actual instruction set (in which case the “optimization” was not exactly for the real machine anyway). Meanwhile Shake and Bake amplifies retargetability, because the same algorithm can be used for a wide variety of instruction set architectures – even provisional ones that are being explored during processor design.

Modeling SHAKE and BAKE as CSPs and the speed of the underlying EGA for solving CSPs make this approach feasible. The ability to generate a pool of candidate solutions and evaluate them quickly allows SAB to investigate multiple trajectories. Although the approach is new and much more testing remains, our results are quite encouraging. We have successfully mapped code to two complex DSPs and several ASIP configurations – all with the same algorithm.

8. Bibliography