6

OASIS: An Optimized Code Generation Approach for Complex Instruction Set PDSPs

Jim K. H. Yu* and Yu Hen Hu
University of Wisconsin–Madison, Madison, Wisconsin

1 INTRODUCTION

A programmable digital signal processor (PDSP) is a special-purpose microprocessor with specialized architecture and instruction set for implementing DSP algorithms. Typical architectural features include multiple memory partitions (on-chip, off-chip, data memory, program memory, etc.), multiple (generally pipelined) arithmetic and logic units (ALUs), nonuniform register sets, and extensive hardware numeric support [1,2]. Single-chip PDSPs have become increasingly popular for real-time DSP applications [3,4]. The newest introductions from several manufacturers offer peak computing power approaching that of supercomputer systems of only a few years ago. However, to fully utilize the available computing power, the software designer must face the difficult task of programming in low-level assembly language.

High-level languages (HLLs) are attractive to software designers because they simplify the task of programming. Unlike assembly language, HLL programs are readable, maintainable, and portable to other processors. All of these features contribute to increase productivity and reduce development cost.

An HLL compiler translates the instructions present in an HLL program into assembly instructions, more easily understood by the processor. Commercially available HLL compilers for PDSPs have existed for many years [5]. Unfortunately, the performance of those compilers is still acceptable only to a few noncritical applications. The reason is twofold. First, the majority of those com-

* Current affiliation: Tivoli Systems, Austin, Texas
Pilers are based on technologies developed with general-purpose applications in mind. For instance, compilation speed is an important issue for general-purpose programs, which can reach millions of lines of code, but completely immaterial to many embedded DSP applications. Second, PDSPs present peculiar architectural features that make good code generation difficult. Genin et al. [6] estimated that assembly codes written by human DSP experts perform 5–50 times faster than those obtained with conventional C compilers. Although the performance of current optimizing compilers has improved significantly, careful manual coding, typically with several iterations of ad hoc fine-tuning, is still the only effective approach for DSP applications with stringent constraints on execution time and/or code size.

In an attempt to diminish the speed and size penalties incurred by programming in HLL, some compilers resort to “external help.” Examples of such external help are the support for in-line assembly language code in the source HLL program and the support for calls to subroutines from application libraries written in assembly language [7,8].

The problem with in-lining is obvious. It contradicts the main purpose of using HLL, which is to spare the software designer the burden of programming in assembly language. Chief among the problems associated with using assembly libraries is their lack of portability. Because assembly libraries are optimized for a given PDSP architecture and instruction set, even a simple PDSP upgrade from the same manufacturer would require the development of new libraries, if they were to fully utilize the features of the new architecture and instruction set. Another problem with using assembly language routines is the inherent compromise between generality and efficiency. For example, let us consider a routine for matrix multiplication. A general approach would pass the size of the matrix as a parameter. However, such a routine is obviously not optimal due to the inevitable overhead caused by loop control instructions. It is not uncommon for optimized application libraries to contain different routines to multiply matrices of size $2 \times 2, 3 \times 3, 4 \times 4$, etc. Furthermore, subroutines require an exact input/output parameter format, which the caller program must satisfy; this may translate into additional stack control overhead.

1.1 High-Level Languages and PDSP

Programmable digital signal processor designers never had HLL in mind; conversely, general-purpose HLL designers never had PDSP in mind. Despite successful attempts in controlled case studies [9], conventional HLLs are not suited for describing the semantics of DSP. They also cannot give the programmer direct access to unique features of PDSPs, such as modulo-addressing register arithmetic, special fixed-point scaling modes, dual data memory access, and fractional data types.
Undoubtedly, the best option is to develop a dedicated HLL for DSP. Although this approach certainly has its advantages [10–12], it would require the DSP programmer to learn yet another programming language. Compounding the disadvantages of adopting a dedicated HLL are the enormous effort in developing the dedicated HLL, its learning curve, and the need to rewrite common DSP routines, which are widely available in one of several general-purpose HLLs [13,14].

The alternative approach is to use an existing general-purpose HLL such as Ada [15], C [16], FORTRAN [17], or Pascal [18]. Among those languages, C is, arguably, the most flexible [12]. It has a sufficiently high-level syntax and yet it supports low-level bitwise operations. Due to its increasing popularity, a vast library of C routines is readily available. Furthermore, many commercial C compilers for PDSPs exist, which allows us to evaluate relative performances. Our HLL choice is further justified by the availability of high-level synthesis front ends [19], which can translate graphical object-oriented signal flow block designs directly into C programs.

By using a conventional HLL to implement DSP algorithms, we are handicapped in more than one way. First, as discussed in the beginning of this section, we are limited by the semantic representational power of the HLL. Second, a language can bias the implementation of an algorithm. Even if we could generate optimal code for a given HLL program, the latter may not be the most efficient implementation of a desired DSP algorithm. We will address the later issue using algorithm transformation techniques, which we will describe in more detail in Section 5.

### 1.2 Structure of a Compiler

The typical structure of a compiler is a front end (FE) feeding a back end (BE), as depicted in Figure 1. The FE processes the HLL source program, performing such tasks as lexical, syntactic, and semantic analysis, and builds an intermediate representation (IR) of that program. The BE then translates the IR into the desired assembly code, also known as code generation. The greatest advantage of such a structure is that if the IR is standardized, $N \times M$ compilers can be built with $N$ FE s and $M$ BEs.

Three important subtasks of code generation are **scheduling,** instruction selection, and resource allocation. Determining the order in which IR operators should be processed is known as scheduling. Determining a valid sequence of assembly instructions to translate a given IR operator is known as instruction

---

*Not to be confused with instruction scheduling, which is a fine-level optimization technique, discussed in Section 2.*
selection. Binding variables to physical resources (registers and memory addresses) is known as resource allocation.

Traditional code generation methods for processors with uniform register architectures usually perform scheduling, instruction selection, and register allocation independently, in separate compilation phases. However, for DSP architectures with multipartitioned memory and nonuniform register sets, in which certain registers and memory blocks are specialized for specific usage, those three phases affect each other [20]. Most importantly, the best translation for a given IR operator is run-time and context dependent. For instance, many modern PDSPs offer...
complex instructions that can accomplish several operations. In certain situations, by taking advantage of instruction side effects, a multiplication can be implemented with shifts and additions, at no cost. Hence, efficient code generation for PDSPs must be context sensitive and must combine scheduling, instruction selection, and register allocation into a single step.

2 BACKGROUND AND RELATED WORKS

The design of an FE is a well-understood problem and many tools, such as LEX, SCANGEN, LLGEN [21], and YACC [22], exist to automate that task. BE design is much more complex because the code generator must address the following issues:

1. Some architectures (especially PDSPs) are extremely nonuniform, making it difficult to design good tools to automate BE construction.
2. CISC (complex instruction set computing) machines (most PDSPs are CISC) have complex instructions with overlapping functionality. In case more than one instruction can accomplish a desired task, which one to use is architecture and context dependent.
3. Architecture-specific constraints may complicate optimizations that are considered machine independent in most general-purpose architectures. For example, common subexpression elimination stores the result of an expression in an attempt to avoid unnecessary recomputations of the same expression. However, in some PDSP architectures, it may be less expensive to recompute an expression than to save its results.*

Attempts to formalize [23,24] and automate [25–29] the process of BE construction have been only partially successful. As a whole, one could say that BE construction still remains an ad hoc problem, especially if the quality of the generated machine code is the primary factor to be considered.

2.1 Code Generation Methods

There are various approaches to code generation. A more detailed discussion of such techniques can be found in classic compiler textbooks such as Refs. 21 and 30. In this section, we will briefly introduce a few relevant methods to build the necessary background for the discussions in the following subsection. The interested reader is referred to the cited literature for further details.

* To a lesser extent, this is also true for other architectures.
2.1.1 Tree Traversal

In this method, the source code is represented by a parse (or syntax) tree. The code generator traverses the tree, usually bottom-up, and invokes small code-generating routines (or macros) for each operator in the tree. The simplicity of this approach has made it a popular code generation method. The main disadvantages of this approach are as follows:

1. Retargeting the code generator to a different architecture requires rewriting all the macros, a time-consuming task.
2. Code quality is directly related to the skill of the macro writer; hence, there is no guarantee that all available instructions on the target machine will be efficiently used.
3. The best sequence of assembly instructions for a given operator is both architecture and context dependent and cannot be captured by predetermined macros. For instance, depending on the operands, a multiplication could be implemented more efficiently as a series of shifts and additions, rather than by a simpler, but often costlier, MPY instruction. Moreover, some PDSPs, such as the TMS320 series, allow the shift to be bundled in many other instructions such as ADD and SUB, effectively providing a zero cost multiplication.

One approach to overcome disadvantage 3 is to enumerate all possible implementations of an IR operator and choose the best one for the current context through an exhaustive search. This method has been investigated by Krumme and Ackley [31]. Their compiler is driven by a set of code tables, each containing all possible (useful) ways to translate the corresponding operator. An interpreter traverses the program tree, one expression at a time, invokes the appropriate code table, and interprets its contents. Krumme’s compiler returns the least expensive among the successful translations as the best implementation for the corresponding operator. In case an operand of an operator is itself an expression, the process recurs.

Despite the exhaustive search formulation, their compiler does not generate optimal codes. Its efficiency is reported to be only “as good as that of other optimizing compilers.” The reason is simple: To overcome the combinatorial explosion of an exhaustive search, the original problem is broken down into subproblems, each corresponding to one statement in the input program. However, the combination of optimal solutions to subgoals does not necessarily yield the global optimal solution [32,33].

2.1.2 Template Pattern Matching

Another method of code generation is that of template pattern matching. In this approach, a set of basic elements (templates) is defined and must completely
describe the IR in question. A sequence of machine instructions is associated to each of these templates. The code generation problem now translates to a covering problem (i.e., finding a set of templates that can completely cover the IR). If a cost is also associated with each template, then we can define optimal code generation as finding the cheapest cover for the IR. Once a set of matching templates is selected, their associated machine instructions form the desired output code. Incidentally, if the IR is a tree, it has been shown that, for a certain class of machines,* a covering of least cost can be found in time linearly proportional to the size of the tree [23]. Hoffmann and O’Donnell [34] and Chase [35] give very efficient algorithms for tree pattern matching.

A directed acyclic graph (DAG) [21] is a generalization of the tree, in which nodes may have more than one parent. Support for multiple parents allows sharing of common subexpressions and, consequently, yields a more compact representation. Figure 2 illustrates the tree and DAG representations for the expression \((A + B) + (A + B)\). The term *acyclic* implies that no cycles are allowed, as the latter may lead to infinite loops. Unfortunately, finding a least cost cover for a DAG is an NP-complete problem [36].

The earliest works on this approach are those of Wasilew [37] and Weingart [38]. Ripken [39] and Johnson [40] also proposed code generation schemes based on this approach. Glanville [41] also used templates as a machine description, but automatically derived a transition table from them, resulting in a faster code generator.

The dynamic programming algorithm used in Twig [28,42] is among the most efficient for template pattern matching in trees. It implements the algorithm proposed by Aho and Johnson [23]; consequently, optimal coverage can be accomplished in time linear with the size of the input tree. However, Twig is a

* The processor architecture assumed by Aho and Johnson [23] has \(n\) general-purpose registers and a finite number of memory locations. All registers are interchangeable, as are all memory locations.
The quality of the generated code is directly dependent on how faithfully the target machine’s instruction set is represented in the Twig rules. Such rules must account for all possible side effects associated with individual instructions, so typical of PDSPs. The optimality of Twig’s solution is heavily constrained on the following: (1) It is assumed that the target machine has a uniform-register architecture with \( r \) interchangeable registers and (2) evaluation is contiguous and over one expression tree. Many PDSP architectures are not uniform register and may not allow optimal contiguous evaluations. Twig does not provide means for high-level optimizations such as common subexpression elimination, although algebraic simplifications can be accomplished by tree rewrites (see Sec. 2.1.3); peephole optimization [43] must be used to handle boundaries of two code sequences. Register assignment and allocation must be done separately by a user-provided routine.

The works of Newcomer [44] and Cattell [26] are of particular interest to us. Although our system is structurally different from theirs, many similarities inevitably arise due to the application of the same conceptual tools, namely heuristic search and means-ends analysis (MEA).* Hence, it is appropriate to conduct a more detailed discussion of these works.

In 1975, Newcomer pioneered the idea of applying MEA [44] in the code generation domain. Probably because of the (lack of) performance of early computer systems, Newcomer (and Cattell) declared MEA “too expensive” and restricted its application in a template generation phase. His work laid the basis on which Cattell later extended and improved. For the purposes of this work and because both Newcomer and Cattell used the same underlying paradigm, it suffices to discuss Cattell’s work. The interested reader is referred to Ref. 45 for an informal but detailed summary of Ref. 44.

Cattell divided the code generation process into a machine-dependent and a machine-independent phase. In the machine-dependent phase, at compiler compile time, the effects of each instruction are described using a register-transfer notation. Then, a template generator creates a finite set of patterns to be used in the code generation phase. For each template, the best sequence of machine instructions is obtained through MEA and heuristic search. This process is performed once for each target machine. In the machine-independent phase, at compile time, a tree pattern-matching algorithm called maximal munching method (MMM) is used. Briefly, MMM tries to cover the input tree recursively, in a top-down fashion, from the root to the leaves. At any node, MMM selects from the available templates the one that can cover (bite off) the largest possible subtree.

* Means-ends analysis is a well-established artificial intelligence technique that takes into consideration the available resources (means) and the goals of the problem (ends) in specifying a plan of action.
Once a template matches a portion of the tree, the sequence of machine instructions associated with that template is emitted as part of the generated code.

Generation of templates is based on a set of axioms, which expresses classical arithmetic and Boolean equivalencies. Rules are used to specify the equivalence of programs (e.g., fetch/store decompositions and sequencing semantics) and idiosyncrasies of the target architecture (e.g., side-effect compensation). These axioms are divided into three classes:

1. **Transformations.** These are the axioms concerned with arithmetic and Boolean equivalence. Transformations are used in conjunction with MEA.
2. **Decompositions.** These axioms are normally concerned with control constructs; they decompose constructs into sequences of other constructs, allowing the search to proceed recursively on subgoals. Decompositions are used in conjunction with a general heuristic search.
3. **Compensations.** These are the axioms concerned with side effects. No search is associated with these axioms.

Cattell’s method has the advantage that the quality of the assembly instruction sequences associated with individual templates does not depend on the skills of a human writer.* Albeit suboptimal, those sequences are generated in a systematic, consistent, and automatic manner. Hence, portability is greatly enhanced. The main drawback is that it does not address the issue of the best sequence of machine instructions being compile-time context dependent. Furthermore, special subtrees not selected to be included in the set of patterns are not analyzed in the search process, resulting in suboptimal codes. Separate pass optimization techniques such as peephole must be applied. In addition, the original MEA (as used by Cattell), being table driven, poses some limitations to code generation. It is simply impossible to enumerate and rank all possible solutions in a table.

In Section 4, we will present a modified MEA that is context sensitive and able to evaluate the best candidates at compile time.

### 2.1.3 Tree Rewrite Systems

A tree rewrite system consists of a set of rewrite rules of the form $A \rightarrow B$, where $A$ and $B$ are tree patterns. Similar to the template pattern-matching method, each rule is associated with a sequence of machine code to be generated and, optionally, a cost. In such a framework, (optimal) code generation translates to solving

---

* In the tree/DAG traversal method, the code-generating routines (or macros) must be handwritten. In Twig, the assembly instructions associated with a given pattern are also handwritten.
the (C) reachability problem: given an input tree T and a goal tree G, determine if there exists a sequence of rewrite rules to rewrite T into G, and if so, obtain one such (the cheapest) sequence.

Pelegri-Llopart and Graham [46] implemented an optimal code generator for expression trees based on rewrite systems. In their paper, they describe a table-driven algorithm to solve reachability based on bottom-up rewrite systems (BURS) theory and submit that BURS-based code generation is faster than one based on Twig.

Wendt [29] developed a code generator, CHOP, based on rewrite rules for DAGs. CHOP reads nonprocedural descriptions of a target machine’s instruction set and of a naive code generator for that machine. It then constructs an integrated code generator and peephole optimizer for that machine. A learning version of the compiler infers the optimization rules as it compiles a training suite and records them for translation into hard code and inclusion into the production version. CHOP is reported to run 30–50% faster than Johnson’s landmark Portable C Compiler [47] and generates comparable code.

2.1.4 Knowledge-Based Systems

Many researchers have proposed providing the code generator with some form of expert knowledge about the processor’s architecture and/or the application domain. One of the first attempts to incorporate human knowledge into code generation is described in Fraser’s dissertation [25]. Fraser designed an expert system specialized to code generation. In his system, the IR is matched against a set of machine-independent rules; the action parts of those rules specify the assembly code to be generated. Retargetability is feasible only among similar processor architectures, because rules are not automatically generated and one must pay particular attention not to write new rules that may clash with existing rules.

Genin et al. [6] reported a system in which, by providing the compiler with knowledge about DSP and the target processor’s instruction set, the generated code performs “5 to 50 times faster than the one produced with conventional C compilers and is comparable to the code generated by DSP experts.” An internal signal flow graph is generated in which each node represents a DSP operation of a specific class. A pattern recognizer identifies and merges low-level nodes into semantically higher-level constructs. Each node is responsible for its own code generation and node activation is determined by a scheduler. The scheduler has knowledge of the global context through a set of if–then rules. The action parts of the rules contain the code to be generated. One of the key reasons for such an impressive performance is that this system requires the use of a specialized DSP HLL called Silage [10].
In DISSIPLE [48], the code generator is composed of two rule-based expert systems; one generates the control flow code and the other the computation code. The use of separate code generators is claimed to be advantageous for portability reasons and to allow the compiler to generate code for both single-processor and multiprocessor systems. Code optimization, however, is deferred until the last step of the compilation process, through conventional peephole techniques.

Kuroda et al. [20] developed a knowledge-based code generator for PDSPs with nonuniform registers and partitioned memory architectures. Such architectures are designed to allow multiple memory accesses in a single instruction. Hence, efficient memory assignment of DSP specific data types is critical to the quality of the generated code. A profiler analyzes the HLL program to obtain statistics about the execution times for instructions with multiple memory access. An access conflict graph, in which nodes represent data values and edges represent the number of conflicts, is constructed from the collected statistics. A heuristic algorithm then assigns variables to the several memory partitions, minimizing the number of conflicts. The code generator first translates the IR into a register-transfer-level representation by binding processor resources to each variable. Then, a recursive pattern-matching process is carried out to cover the entire IR. Patterns are if–then rules in which the then part is associated with instructions to be generated (including register allocation) and the if part is associated with a (sequence of) register-transfer operation(s) that realize the then part. By checking the number of additional steps required to generate a code, the code generator can select the best code among the alternatives stored in the rule base. Unfortunately, no quantitative results are presented, except the authors claim that their compiler “can generate almost the same code as programmers would have done by hand.”

Azaria and Dvir [49] described an expert-system-based compiler for a special-purpose array processor. Because their proposed system (OC) can handle only the four elementary mathematical operations (+, −, º, /), it was dedicated to solving linear constant systems and other problems that can be totally defined by a series of matrix operations. In such a restricted environment, expert knowledge was shown to be very effective. OC’s high optimization performance is accomplished mainly through algorithm transformation. By consulting the rule base, OC identifies and eliminates redundancies in the source algorithm.

2.2 Optimizations

An optimizing compiler is one that applies optimization techniques such that the resultant machine code is faster and/or smaller than that produced by conventional compilers. Perhaps the most important tenet about optimizations is that they aim at improving code, rather than actually attaining optimal code.
Optimizations can be classified into two categories, based on the type of information that is used: static and dynamic optimizations.

2.2.1 Static Optimizations

Static optimizations rely on compile-time analysis information to alter the program. Static analyses are inherently limited and imprecise because many aspects of run-time behavior are undecidable at compile time. Nonetheless, static optimizations are relatively easy to perform, with little overhead in compile time and good potential for gains at run time.

Among the myriad of ingenious static optimization techniques implemented in current optimizing compilers, the most interesting include the following:

1. Algebraic simplification and rearrangement: Replacing expressions with simpler, equivalent expressions, or rearranging expressions to facilitate constant folding.
2. Branch-tail merging: Combining instructions common to branches that jump to the same location.
3. Code motion (hoisting): Moving instructions common to the beginning of two branches of a conditional jump to a location before the jump.
4. Common subexpression elimination: The compiler computes the value of expressions that always yield the same result, saves the result as a temporary value, and uses that value instead of recomputing the subexpression each time it encounters them.
5. Constant folding: Replacement of arithmetic and logical operations by the constant to which they evaluate at compile time.
6. Constant propagation: Changing variable references to constants, if the variable has a constant value.
7. Dead (unreachable) code removal: Removal of code with no path. Other optimizations generally create dead code, not the programmer.
8. In-lining: In-line expansion of a function to save the overhead of the call, parameter passing, register saving, stack adjustment and value return.
9. Instruction scheduling: Rearranging instructions sequence to avoid stalling the instruction pipeline; includes overlapping CPU and FPU instructions and inserting useful instructions between data-dependent instructions that stall the pipeline.
10. Loop rotation: Rotation of the controlling expression of FOR and WHILE loops to the bottom of the loop. If the loop executes more than once, this optimization eliminates the initial test.
11. **Loop unrolling**: Using multiple copies of a loop’s body statement instead of incrementing and checking the loop induction variable.

12. **Redundant code elimination**: Elimination of code sequences producing no real side effects; usually result from other optimizations.

13. **Register coloring**: Optimizes register usage throughout a function or routine, such as keeping local variables in registers at all times. Similar to coloring a map where no two adjacent areas can have the same color, the algorithm uses data flow (lifetime) analysis for each variable, mapping variables without overlapping lifetimes to the same register.

14. **Short-circuit (logical expression) evaluation**: Converting expressions containing AND, OR, and NOT operators into a series of conditional jumps.

15. **Strength reduction**: Replacing an operation with equivalent but less expensive (in size and/or speed) operations; for example, replacing multiplication with combinations of shifts and adds.

16. **Switch statement optimization**: Choosing among different code generation methods for switch statements based on code size or speed. Methods include linear test and branch, jump table, binary search, and parallel tables.

Few commercially available optimizing compilers for PDSPs have been introduced [7,8,50]. To varying degrees, they all perform static optimization techniques such as those listed above. The biggest challenge for an optimizing compiler is to apply optimizations in such an order that each optimization can uncover all possibilities for additional optimization opportunities and effectively exploit them. More importantly, the compiler must ensure that one optimization does not undo the results of previous optimizations. An interesting approach is that used in Benitez’ Very Portable Optimizer (VPO) [51]. VPO uses an iterative approach: Each optimization is applied in separate phases; previously executed phases are reinvoked whenever new possibilities arise.

Static optimizations can be performed in three stages of the compilation process, as illustrated in **Figure 3**. Typically, optimizations performed in the FE or on the IR are machine independent. Their main purpose is to ease the code generator’s job rather than to improve code quality. The BE has access to target-processor information, hence, the code generator can produce both machine-dependent and machine-independent optimizations. Certain optimizations, such as instruction scheduling and register coloring, are intended for postgeneration. These are inherently machine dependent.

Most commercially available optimizing compilers also perform peephole optimizations [24,43,52,53]. Peephole optimizers work on the assembly code by looking at a few (typically two or three) instructions at a time and attempt to
replace that sequence of instructions by another sequence, semantically equivalent but more efficient. The fundamental drawback of this technique is that it is localized in nature. Increasing the window size (looking at more instructions) increases computational complexity exponentially without much improvement in performance.

2.2.2 Dynamic Optimizations

Dynamic optimizations encompass all those optimizations that use run-time analysis to alter the program. They can be further subdivided into context-dependent and data-dependent optimizations. These two categories are not exclusive and can complement each other.

**Context-Dependent Optimizations.** The major shortcoming of static optimizations is that the compiler cannot decide which, among a large set of possible optimizations, will be beneficial and which will hurt code quality. A simple approach is to subdivide the set into smaller sets and classify them according to their generality (or innocuousness) and let the user decide when and which sets to apply. There are several problems with this approach. First, determining the generality of an optimization is a nontrivial task. As we will show in Section 5, an optimization such as common-subexpression elimination, which is typically considered general and therefore safe to perform whenever possible, is actually architecture and context dependent in the presence of run-time information. Second, the user must go through a tedious analysis of the generated code to determine which optimizations were the culprits for the code’s poor quality and recompile with those optimizations disabled. What is worse, an optimization may be beneficial in one part of the program and harmful in others. Third, the order in

---

**Figure 3** Three different stages of compilation where static optimizations can be applied: (a) before, (b) during, and (c) after code generation.
which optimizations are applied is important and there is no easy way to determine the best order. More importantly, the best order may depend on the input program.

Context-dependent optimizations or CDAT, as we will refer to them in Section 5, use run-time analysis to determine the status of all resources, such as registers, various memory partitions, and status flags to determine if a particular optimization is advantageous to implement.

As we will see in Section 5, many interesting issues arise in interpretive code generation. We will discuss each one of them in detail and show that our heuristic search methodology for code generation can solve them quite naturally.

Data-Dependent Optimizations. Alternatively, one could consider optimizing a program based on the data that it is supposed to process. Because the actual data are unknown until run time, data-dependent optimizations and, consequently, code generation must be deferred until run time. The generated code, thus, contains instructions to emit code at run time. Many approaches have been proposed to reduce the overhead imposed by run-time code generation (RTCG), including the use of templates [54–56] and the use of a more general intermediate representation [57].

Interesting works on RTCG include that of the SELF group [92,93,58] and FABIUS from Carnegie-Mellon [94]. A good summary of other work on RTCG can be found in the technical report by Keppel et al. [56].

Run-time code generation has a high degree of flexibility in the sense that it can optimize the code for a particular set of data, possibly even for different iterations of the same loop. On the other hand, it is difficult to implement and the overhead added to the run time is recovered only for large input sizes. Many DSP-embedded applications value code size over execution speed,* which is substantial detriment to using RTCG methods.

An alternative approach to RTCG is to utilize run-time profiling information to optimize the program in subsequent compilations. University of Washington’s SELF compiler has had some success using dynamic profile information to guide compile-time optimization of object-oriented code [59,60,93]. This approach would eliminate the code generation overhead in the executable code, at the expense of longer compilation times and complexity in the profiling analysis. The major shortcoming, however, is that for meaningful results, the input data must be either very predictable or known exactly.

* The available memory size is limited. In addition, the assumption that a small code will also execute fast is often valid.
2.3 Summary and Conclusions

In this section, we have presented four code generation methods: tree/DAG traversal, template pattern matching, tree rewrite, and knowledge based. We also surveyed many systems and research works within each of these methods and discussed their intrinsic advantages and disadvantages. None of the reviewed methods alone can generate code whose quality is truly comparable to that of handwritten code by DSP experts.

We also surveyed static and dynamic optimization techniques. We discussed why static optimizations are necessarily imprecise. Dynamic optimizations can be data and context dependent. The former can potentially produce better code than the latter. However, because the data are unknown until run time, the optimizations (and code generation) must be deferred to run time. This poses a heavy load on the executable code, because it now has to generate its own code. More importantly, RTCG increases the size of the executable code, a penalty not affordable by many embedded applications. A more suitable approach is that of context-dependent optimizations, which can produce very efficient code at the expense only of compilation time.

In the next section, we present our proposed approach, which combines tree/DAG traversal, template pattern matching, and knowledge based with other artificial intelligence techniques and context-dependent optimizations to attain the desired handwritten code quality.

3 OVERVIEW OF OASIS

Our formulation of code generation is a hybrid of the tree/DAG traversal, template pattern-matching, and knowledge-based approaches (introduced in Sec. 2). From the highest-level view, the algorithm, which we call OASIS (Optimized Allocation, Scheduling, and Instruction Selection), works as illustrated in Figure 4. First, a DAG is built from the input program. Next, the DAG is augmented to include flow-of-control information, followed by scheduling of each node in the augmented DAG. This initial scheduling, however, does not determine the actual order in which nodes are covered. For the latter, a more complex heuristic algorithm is required because the quality of the generated code is directly related to the order in which nodes are covered. At each iteration, all coverable nodes are collected; each of those coverable nodes is evaluated and the least expensive node is selected and covered. If more than one node share the least expensive cost, then the node with the lowest initial schedule is selected. Node coverage

* We note the distinction between the terms optimal and optimized. The first refers to the absolute best. The latter is a frequently used term by compiler vendors, implying that optimization techniques have been applied and the result may or may not be optimal.
involves template pattern matching, whereas node evaluation involves heuristic search; both node coverage and node evaluation also involve means-ends analysis and hierarchical planning, which we will examine in more detail in the next section.

Unlike the approaches of Newcomer [44] and Cattell [26], there is no separate compiler compile phase. All analysis is done at compile time. The advantages of such an approach include the potential to achieve optimal code,* at the expense of longer compilation time and larger memory requirements.

* Due to the use of heuristics, an optimal solution cannot be guaranteed. However, for a sufficiently small DAG and given sufficient computational resources, if an optimal solution falls within the solution space defined by the heuristics, OASIS will find it.
As we explained in Section 1, many embedded DSP applications can justify the costs of a more complex code generation algorithm and we will not concern ourselves with such issues as compilation time and memory requirements of the algorithm. Our main goal is to provide a means to obtain the best possible machine code. Nonetheless, the entire task must be realizable within practical limits of time. We will show in the next section how the application of artificial intelligence (AI) tools can reduce an essentially NP-complete problem into one that executes in polynomial time with the DAG size, albeit exponentially in the search depth $K$. Because $K$ is user defined, our algorithm provides the flexibility to match computational resources with desired code quality. We will show in Section 6 that very good results can indeed be obtained with relatively small $K$ and surprisingly short compilation times.

Figure 5 presents the structure of the entire system. The five gray blocks comprise the back end, where code generation occurs. The three lighter gray blocks are machine independent, whereas the two darker gray blocks are machine dependent. Porting this code generator to a new architecture then requires only a new rule base and a new instructions library.

### 3.1 The Front End

First, let us define a small example, which we will use throughout this section to illustrate specific issues. Consider the following fragment of C code:
Example 1

$S1: \quad x = b \ast c;$
$S2: \quad y = x + b \ast c + d \ast e;$

In our current implementation, the front end (FE) is adapted from lcc compiler [61]. The FE takes an ANSI C program as input and generates a linearized DAG as the intermediate representation (IR). Because the linearized form is quite difficult to visualize, we will use a graphical representation of the FE’s output for Example 1, as shown in Figure 6. Nodes of type INDIR represent the fetch of a variable value from a resource, and nodes of type ASGN write a new value to the variable stored in a certain resource. The translation from the linearized form to the graphical form loses some information, namely control and data dependencies. In Example 1, it implies that statements $S1$ must be executed prior to $S2$. However, in Figure 6, there is nothing that indicates that node 1 should be covered before node 12. In the next subsection, we show how that information can be recovered, as well as discuss other pertinent issues related to the IR.

3.2 The Intermediate Representation

Many static optimization techniques can be applied at the IR level, such as constant folding and relocation of invariant loop expressions [62]. Static algorithm
transformation techniques can also be applied at the IR level to transform the input program into an equivalent representation that better matches the target processor's architecture (see Sec. 2.2). Our formulation of code generation calls for context-dependent transformations, which must be deferred until compile time, at the BE. Hence, the IR preprocessing block only augments the DAG with control and data dependency arcs, as well as scheduling information, which are needed by the BE.

3.2.1 Data Dependencies

A data dependency occurs between two statements when both reference the same memory location or access the same physical register. There are three kinds of data dependencies [63,64]:

1. True dependency (also called flow dependence or read-after-write hazard): characterized by

   $$\begin{align*}
   S_1: & \ x = \ldots \\
   S_2: & \ldots = x
   \end{align*}$$

   The value of \( x \) at statement \( S_2 \) depends on the value assigned to it at \( S_1 \); hence, the assignment in \( S_1 \) must be executed before the use of \( x \) in \( S_2 \).

2. Antidependency (also called write-after-read hazard): characterized by

   $$\begin{align*}
   S_1: & \ldots = x \\
   S_2: & x = \ldots
   \end{align*}$$

   A new value cannot be assigned to \( x \) until \( x \)'s current value has been used; hence, the use of \( x \) in \( S_1 \) must be executed before the assignment in \( S_2 \).

3. Output dependency (or write-after-write hazard): characterized by

   $$\begin{align*}
   S_1: & x = \ldots \\
   S_2: & x = \ldots
   \end{align*}$$

   The final value that \( x \) assumes depends directly on which statement is executed last. The order in which the statements appear in the source program must be preserved by the scheduling algorithm. In this case, the assignment in \( S_1 \) must be executed before the assignment in \( S_2 \).

   As the name implies, true dependencies are the only real dependencies; a variable required by the current statement must have its value evaluated by a previous statement. Both antidependence and output dependence can be eliminated by variable renaming [65,66]. Variable renaming usually increases the re-
quired memory for the program. Because our methodology supports all three types of data dependency, variable renaming is unnecessary.

The algorithm that follows provides a systematic method for introducing the necessary data dependency arcs to the DAG:

```
AddDataDependencyArcs(DAG) {
    for (every node i in DAG that is of type ASGN or INDIR) {
        for (every node j • i in DAG that is of type ASGN or INDIR) {
            if (no arc exists between nodes i and j) {
                Dep Type = DetermineDependencyType(DAG, i, j);
                case (DepType) {
                    TrueDep: add arc from node i to node j;
                    AntiDep: add arc from node j to node i;
                    OutputDep: if (i > j) add arc from node i to node j
                              else add arc from node j to node i;
                };
            }
        }
    }
}
```

Nodes of type INDIR represent the fetch of a variable value from a resource, and nodes of type ASGN write a new value to the variable stored in a certain resource. Hence, data dependency arcs occur only among INDIR and ASGN nodes. However, not all INDIR and ASGN nodes need a data dependency arc, only those that falls into a true, antidependency or output dependency category. Figure 7 illustrates one such case in which no data dependency arc is neces-

![Figure 7](image)

**Figure 7** DAG representation of \( x = x + \ldots \).
sary between ASGN and INDIR operators. The dependency relationship in S1 is uniquely defined: The ASGN operator (node 4) must be scheduled after the + operator (node 3), which, in turn, must be scheduled after the INDIR operator (node 2).

3.2.2 Control Dependencies

Intuitively, a control dependence occurs from statement S1 to statement S2 when S2 is executed only if S1 produces a certain value [64,67]. This type of dependence occurs, for example, when S1 is a conditional statement and S2 is to be executed only if the condition evaluates to true. To illustrate when control dependencies might occur, let us consider Example 2:

Example 2

S1: if (a) x = 1;
S2: else x = 2;
S3: y = x;

The linearized DAG generated by the FE is as follows:

```
node#3 ADDRLP count = 1 a
node#2 INDIRI count = 1 #3
node#4 CNSTI count = 1 0
node’1 EQI count = 0 #2 #4 2
node#6 ADDRLP count = 1 x
node#7 CNSTI count = 1 1
node’5 ASGNI count = 0 #6 #7
node#9 ADDRGP count = 1 3
node’8 JUMPV count = 0 #9
2:
node#12 ADDRLP count = 1 x
node#13 CNSTI count = 1 2
node’11 ASGNI count = 0 #12 #13
3:
node#16 ADDRLP count = 1 y
node#18 ADDRLP count = 1 x
node#17 INDIRI count = 1 #18
node’15 ASGNI count = 0 #16 #17
```

In this linearized DAG, node’5 ASGNI count = 0 #6 #7 reads as node 5 is the root of a subtree, with no arcs pointing to it (count = 0), its type is ASGNI.
and it has arcs pointing to two other nodes, node 6 and node 7.* The same DAG, in a more legible graphical form is shown in Figure 8.

The lcc compiler transforms *if–then–else* structures into a one-dimensional description with GOTOs (JUMPs) to determine ordering. When a two-dimensional DAG is built from such description, the ordering information is lost. For example, in the linearized DAG, nodes 11 to 13 precede nodes 15 to 18. A JUMP in node 8 brings us to the subtree labeled 3, but no JUMP is necessary at the end of the subtree labeled 2 because it is assumed that the subtree labeled 3 follows unconditionally. However in the graphical representation of the linearized DAG, shown in Figure 8, subtrees are spread around, with nothing to indicate the covering order. One way to recover the precedence information is to augment the DAG with control dependence edges. Such edges are processed by the IR Preprocessing block just like any other edge. Hence, if a node M must be covered before a node N, an edge must be added from N to M. This forces a partial scheduling order more in line with the one meant by lcc. The resultant augmented DAG is shown in Figure 9.

### 3.2.3 ALAP and ASAP Schedulings

Once the DAG has been augmented with data and control dependency arcs, the next step is to assign an initial schedule to each node in the DAG. We use an

---

* For a complete description of lcc’s syntax, see Ref. 61.
as-late-as-possible (ALAP) scheduling. In an ALAP schedule, if node M must be covered before node N, M is scheduled as close to N’s schedule as possible. The rationale for using ALAP is as follows: If M must be covered before N, then there must be an arc from N to M. If it is a data dependency arc, then N depends on M’s data and executing M immediately before N makes the least commitment of resources, yielding the positive effect of reducing spill instructions. If it is a control dependency arc, then any scheme that schedules M before N can be applied and ALAP provides a valid initial guess. Note that the actual coverage order will be determined by a more thorough node analysis, as we will see in the next sections.

The algorithm to perform as-soon-as-possible (ASAP) scheduling is as follows:

\[
\text{ASAP} (\text{DAG}) \{ \\
\quad \text{node[root].schedule} = 0; \\
\quad \text{N = set of all nodes of the DAG;} \\
\quad p = 1; /* schedule */ \\
\quad \text{while (N not empty) } \{ \\
\quad \quad \text{for (every node j in N) } \{ \\
\quad \quad \quad \text{if (all parents of node j have been scheduled) } \{ \\
\quad \quad \quad \quad \text{temp[j] = p; } \\
\quad \quad \quad \quad \text{N = N-\{j\}; } \\
\quad \quad \quad \} \\
\quad \quad \quad \text{else temp[j] = -1; } \\
\quad \quad \}; \\
\quad \quad \text{for (i = 1; i < number of nodes; i++) } \{ \\
\quad \quad \}
\]
if (temp[i] != -1) {
    node[i].schedule = temp[i];
};
}
}
p++;
}

ASAP is essentially the opposite of ALAP, as it tries to schedule nodes immediately after their parents have been scheduled. The ALAP scheduling is derived from the ASAP scheduling and its algorithm is as follows:

\[
\text{ALAP(DAG)} \quad \text{(DAG)}
\]

\[
\begin{align*}
&\text{Reverse the directions of all edges in DAG;} \\
&\text{Assume the '"root'" is '"output'" and vice-versa;} \\
&\text{ASAP(DAG);} \\
&n = \text{largest schedule #;} \\
&\text{for (all nodes j) } \\
&\quad \text{node[j].schedule} = n-\{\text{node[j].schedule}\} + 1; \\
&\end{align*}
\]

This ALAP initial scheduling has three purposes. First, it provides a basis for the actual node coverage ordering. At each iteration, the node evaluation functions (described in Section 4.1.3) evaluate the effective cost of all expandable nodes. The least expensive node is then selected for coverage. Second, it is common for more than one node to share the minimum effective cost. In that case, the decision is made based on the initial scheduling—the node with the lowest initial schedule is selected. It is still possible, however, that more than one node share both the minimum effective cost and initial schedule number. In that case, an arbitrary decision is made: The node with the lowest node number is selected.

As we will see, node evaluation is the single slowest component of the whole algorithm. OASIS allows the user to specify a limit, B, on the number of nodes to be evaluated. Again, the initial schedule plays an important role. Coverable nodes are sorted in increasing initial schedule and only the first B nodes are evaluated.

The scheduled DAG for Example 1 is shown in Figure 10. In Example 1, variable x in S2 depends on the value assigned to it in S1. This data dependency is indicated by the arc from node 12 to node 1, which causes node 12 to be scheduled after node 1.

### 3.3 The Back End

Code generation is performed in the BE, which comprises the last five blocks in Figure 5. There are three processor-independent modules—search engine, algo-
Figure 10  Augmented and scheduled DAG for Example 1, as received by the BE.

Algorithm transformation, and means-ends analysis—and two processor-dependent modules—rule base and the instructions library. This arrangement makes porting the code generator straightforward. All that is required is a new rule base, with heuristics, rules of thumb, or any expert information about the new architecture, and a new library of templates representing the new instruction set. The BE’s main functional block is the search engine, which is assisted by the rule base and the instructions library modules and interacts with the algorithm transformation and means-ends analysis modules.

Each of the above functional blocks will be addressed in detail in Section 4.

3.4 Summary and Conclusions

In this section, we presented an overview of a proposed compiler. It is essentially an FE feeding a BE type of structure, as introduced in Section 1. FE generation is well defined using models such as context-free grammars; many tools exist to automate that task. Our FE is adapted from that of lcc [61]. Unlike FE design, good BE design remains more of an art than science and gives much room for ad hoc approaches.

An IR preprocessing module is introduced between the FE and the BE. That module transforms the IR generated by the FE into a form more suitable for the BE. This section discussed in great detail the tasks involved in the IR preprocessing module.
In this section, we briefly introduced our formulation of code generation as a heuristic search problem. Detailed discussion of our formulation of code generation will be conducted in the following sections.

4 CODE GENERATION IN OASIS

In the previous section, we discussed in detail the FE and the IR preprocessing modules. In this section, we will focus our discussion on the BE, where code generation is implemented. We will discuss in detail how artificial intelligence techniques such as means-ends analysis and hierarchical planning are combined in the heuristic search implementation. We will also conduct a time-complexity analysis to ensure that the entire algorithm is bounded.

4.1 The Search Engine

The search engine builds two types of heuristic search trees, illustrated in Figure 11. Each state (search node) in these search trees contains, among other information, the remaining DAG to be covered and the status and contents of all physical resources of the target processor. The latter information is encoded into a resource table (RT).

An instructions library containing all available instructions of the target processor assists the search engine. In the instructions library, one assembly instruction or a small sequence of assembly instructions can define a template pattern. Each template is described by a set of preconditions and a set of postconditions. Expanding a search node into a successor node is achieved by applying an instruction or matching a template* (see Section 4.1.1). A template matches if its preconditions can be fulfilled by the current node’s RT. Postconditions represent the effects of matching the template and determine the contents of the successor’s RT. In the following representation, we illustrate how the template MPY_dma can be represented. Note that we only indicate which resources hold operands and which are affected by the results of the instruction; we specify neither what kind of operation it performs nor the value of its results. Because templates are not linked to any particular operation, they could be considered for coverage of any IR operator, as long as the required rules are satisfied.

The search engine is also assisted by a rule base, which encapsulates expert knowledge about the code generation domain in the form of heuristics, context-dependent rules and architecture-dependent constraints. In the following simpli-

* Hereafter, we will use both terms interchangeably.
Figure 11  A heuristic search tree with depth \( K = 3 \). At each level \( p \), heuristics are used to expand a node into its most promising successors. The search engine supports two kinds of trees: (a) Each successor is associated with a node cost and only the least expensive successor is selected for expansion at level \( p + 1 \); this kind of tree is used at the topmost level (see Fig. 4). (b) All promising successor nodes are expanded at level \( p + 1 \) and the node cost at level \( p \) is the sum of the expansion cost at level \( p \) and the minimum node cost at level \( p + 1 \); this kind of tree is used to evaluate the node costs of the tree in (a).

A predefined representation of template MPY dma, text within brackets is treated as a comment:

\[
\begin{align*}
(o & = \text{operand}) \\
(l & = \text{left operand}) \\
(r & = \text{right operand}) \\
(s & = \text{result}) \\
(n & = \text{not affected}) \\
\text{(Instruction)} & : \text{MPY dma} \quad (PR \leftarrow TR \ast DM) \\
\text{(Cost)} & : 1 \\
\text{(Resources)} & : PR \ TR \ AC \ DM \ PM \ ARP \ AR \\
\text{(PreCond)} & : n \ oi \ n \ or \ n \ n \ n \\
\text{(PostCond)} & : s \ o \ n \ o \ n \ n \ n
\end{align*}
\]
As shown in the foregoing, each template has a field `Rules`, which contains all the rules from the rule base that are relevant to the current instruction. Because all rules listed in field `Rules` must be satisfied before a template can match, this field can also be used to differentiate two templates that affect the same set of resources identically (i.e., share the same preconditions and postconditions (more on rules in the next section).

Returning to Figure 11a, if a template matches, it expands the current search node into a new search node. The latter contains a new DAG, in which the covered portion is removed. The current search node is expanded into as many successors as there are matching templates. Each successor node is evaluated (see Section 4.1.3) and the least expensive one is selected for the next iteration.

Each search node also has a pointer to its parent and a field `Instructions` to store all the templates that matched during its expansion process. Once a goal node (a node in which the entire DAG has been covered) is reached, the pointers to parents are traced backward from the goal node. The contents of the `Instructions` fields along that path are collected to form the generated code.

### 4.1.1 Modified Means-Ends Analysis

Procedure `ExpandNode` encapsulates a technique known as means-ends analysis (MEA) [68] to expand a search node and return the cost of that expansion. Briefly, MEA works as follows. First, the difference between the preconditions of the candidate template `T` and the RT of the current node is established. This difference is passed to procedure `Apply`. If the difference is nil, `T` is immediately applicable and `Apply` expands the current node with `T`. Otherwise, procedure `Reduce` is invoked to reduce that difference. `Reduce` calls `Apply` with the new, hopefully smaller, difference. In the following algorithm for procedure `Apply`, `Apply` applies template `T` on node `X` to expand it into the successor node `Y`:

```plaintext
Apply(SearchNode X, Template T, SearchNode Y) {
    D = Difference(X, T); /* evaluate difference between X→RT and T→Preconditions */
    if (D is nil)
        Create Y(X, T, Y);  /* create output Y by modifying X→RT and X→DAG to reflect the effects of applying T to X*/
    else
        Reduce(X, T, D, Y); /* reduce difference between X and T, and generate output Y*/
}
```

To reduce a difference, the original MEA technique uses a connection table, which lists all possible candidate operators, ranked by order of efficacy, that can
reduce each type of difference. The first problem is that such a table would be impractically large for our application. Any kind of context-insensitive, predetermined action would be a detriment to the goal of optimized code generation. To preserve the context-sensitive requirement, each difference, under all possible contexts, would have to be listed as a different entry. As for the second problem, ranking operators is a nontrivial and error-prone task, given the number of templates to be considered. Our version of MEA evaluates, at compile time, all relevant templates in the instructions library as to their potential to reduce the current difference. A profiling of the OASIS execution revealed that this on-line evaluation represents less than 1% of the total computation time.

In a straightforward implementation, MEA can lead to a dead end or to circular loop situations. Reduce may return a difference viewed by its evaluation function as smaller than the original difference and, nonetheless, when that difference is passed to Apply, it may not be further reducible, or when further reduced, it is identical to the original difference. In an analogy to game-playing programs, it is necessary to look ahead several moves to discover that a seemingly bad move is in fact the best one.

### 4.1.2 Hierarchical Planning

To address the circular cycle problem, we modify MEA by supplementing it with hierarchical planning (HP). HP is a technique introduced by the ABSTRIPS system [69], which has been shown to generate very little or no backtracking [70]. Korf [33] showed that by using an abstraction hierarchy, it is possible to reduce an exponential-time problem to a linear-time complexity. We have found that two abstraction levels suffice in the instruction selection domain. Once a difference is established, it becomes the goal for procedure Reduce. A plan (sequence of instructions that accomplishes a goal) is sought at the higher abstraction level. At that level, the candidate instruction’s preconditions are ignored. Hence, the best instruction is the one that can bring the current RT closer to the desired goal, independent of its applicability. Once a plan is obtained, its implementation is attempted at the lower abstraction level, in which all preconditions are considered and must be satisfied. In the following algorithm for procedure Reduce, Reduce generates a plan of action to reduce the difference between template T and node X and applies each component of the plan until the difference is completely reduced:

```plaintext
Reduce(SearchNode X, Template T, Difference D, SearchNode Y) {
    Plan = GeneratePlan(X, T); /* generate a list of templates, which represents a plan of action */
    while (D is not nil and Plan is not empty) {
        P = First template in Plan;
```
Apply(X, P, Y);
D = Difference(Y, T);
Plan = Plan−P;
);
);

When a component of a plan is not applicable, the process recurs; subplans are sought to fill the gaps between components in the original plan. When developing subgoals, two important issues must be considered:

1. If more than one instruction is necessary to completely reduce a difference, in which order should those instructions be applied?
2. It is possible that the difference is completely reduced before all components of the plan are applied.

To illustrate both issues, let us consider statement S2 in Example 3:

Example 3

S2: \( y = x + b \times c + d \times e; \)

Suppose instruction ADD is being considered for covering the subtree \((x + b \times c)\). ADD’s precondition requires that \(x\) be in the accumulator and \((b \times c)\) in data memory. Let us assume that the subtree \((b \times c)\) was covered by MPY and that the current RT indicates that \(x\) is in data memory and \((b \times c)\) is in register P. Consequently, ADD is not immediately applicable and Reduce is called with two subgoals:

Subgoal 1: move \(x\) from data memory to accumulator
Subgoal 2: move \((b \times c)\) from register P to data memory

The first issue is to determine which subgoal to pursue first. To address it, a new heuristic search is created. In other words, ExpandNode is recursed with all possible orders of the components of the plan and the least expensive order is returned as the best plan. The first choice is to move \(x\) and then move \((b \times c)\). However, no single instruction in the instructions library can move a value from register P directly to data memory. The best plan for subgoal 2 is to move \((b \times c)\) from register P to the accumulator and then from the latter to data memory. The resulting code would be as follows:

Subgoal 1:

\[
\text{LAC } x \quad ; \text{load accumulator with } x
\]

Subgoal 2:

\[
\text{SACL } \text{temp} \quad ; \text{store } x \text{ in location temp in memory}
\text{PAC} \quad ; \text{move } (b \times c) \text{ to accumulator}
\]
SACL y ;move \((b \times c)\) to location \(y\) in memory
LAC temp ;move \(x\) back to accumulator

The alternative is to move \((b \times c)\) first and then \(x\). The resulting code would be as follows:

**Subgoal 2:**
PAC ;move \((b \times c)\) to accumulator
SACL y ;move \((b \times c)\) to location \(y\) in memory

**Subgoal 1:**
LAC x ;load accumulator with \(x\)

The latter ordering is clearly less expensive and is returned as the better plan.

The second issue comes to play when the plan is executed. Note that after instruction **PAC** is applied, one operand to **ADD**, \((b \times c)\), is in the accumulator, whereas the other operand, \(x\), is in data memory. **ADD**’s prerequisite requires exactly the opposite, but because \(+\) is a commutative operator, **ADD** is applicable at this point. Hence, only the first component of the plan is used and the rest is discarded.

4.1.3 Minimizing Search Space

Full search involves processing \(O(B^K)\) nodes, where \(K\) is the depth of the search and \(B\) is the average branching factor. Hence, there are two obvious ways to reduce the combinatorial explosion: reduce \(K\) and reduce \(B\).

In our approach, \(K\) determines the look-ahead factor and is a user-defined variable. The larger the \(K\), the more accurate is the result returned by the evaluation functions, at the expense of exponentially increasing computation time.

To reduce the effective branching factor \(B\), we apply domain knowledge and heuristics in three different ways:

1. Decide which node(s) can be pruned from the search tree. Our approach is to sort the list of expandable nodes according to increasing initial schedules and expand only the first \(B\) nodes (in OASIS, \(B\) is a user-defined variable).
2. Decide which node to expand next. As we discussed earlier, ALAP scheduling provides a starting point, but it is not sufficiently accurate for our purposes. We use look-ahead to evaluate all \(B\) candidates (detailed next) to determine their actual cost. Only the least expensive node is selected to be expanded (see Fig. 11a).
3. Decide which successor(s) to generate during node expansion. Here, domain knowledge is encoded into rules to eliminate unreasonable can-
didate templates, thus expanding a node into only the most promising successors.

Evaluation Functions. As discussed in Section 4.1, the search engine implements two kinds of search tree. OASIS uses two evaluation functions to determine the cost of a node, one for each kind of tree.

In Figure 11a, the basic idea is to expand only the most promising node. The term most promising connotes minimum cost, as defined by a cost function. Because only one node is expanded, the evaluation function must be very accurate. To accomplish the desired accuracy, we perform look-ahead. Hence, to evaluate the cost of a node \( N \), OASIS defines a heuristic function that is a weighted sum of various attributes:

\[
NC_1(N) = w_1 I(N) + w_2 G(N) + w_3 F(N) + w_4 NC_2(N, K)
\]  

where \( NC_1(N) \) is node \( N \)'s total node cost, \( I(N) \) is the number of templates that matched along the minimal path from the start node up to and including the current node \( N \), \( G(N) \) is the total cost associated with those templates, \( F(N) \) is an estimated cost from \( N \) to a goal, and \( NC_2(N, K) \) is a \( K \)-step look-ahead node cost, computed by the second evaluation function. Currently in OASIS, \( w_1 = w_2 = w_3 = w_4 = 1 \) and \( F(N) \) is a function of the number of operators remaining to be covered in the DAG.

The second evaluation function is computed by a \( K \)-step recursive call to procedure \texttt{ExpandNode}. \( K \) is a user-defined value that tells \texttt{ExpandNode} how many levels to look ahead in evaluating a node’s cost. Each recursion decrements \( K \) by 1, as one can observe in Figure 11b. When called with a node \( N \) as parameter, \texttt{ExpandNode} expands \( N \) into its successors \( S_i \) and returns \( N \)'s node cost. The node cost of a node \( N \), with a \( K \)-step look-ahead, is defined by

\[
NC_2(N, K) = EC_2(N) + \min_{S_j \in \text{BC}} \{ NC_2(S_j, K - 1) \}
\]

where \( NC_2(N, 1) = EC_2(N) \)

The second evaluation function is computed by a \( K \)-step recursive call to procedure \texttt{ExpandNode}. \( K \) is a user-defined value that tells \texttt{ExpandNode} how many levels to look ahead in evaluating a node’s cost. Each recursion decrements \( K \) by 1, as one can observe in Figure 11b. When called with a node \( N \) as parameter, \texttt{ExpandNode} expands \( N \) into its successors \( S_i \) and returns \( N \)'s node cost. The node cost of a node \( N \), with a \( K \)-step look-ahead, is defined by

\[
NC_2(N, K) = EC_2(N) + \min_{S_j \in \text{BC}} \{ NC_2(S_j, K - 1) \}
\]

where \( EC_2(N) \) is the expansion cost of node \( N \), given by

\[
EC_2(N) = w_1 T(N) + w_2 C(N)
\]

where \( T(N) \) is the number of templates that matched in the process of expanding \( N \) and \( C(N) \) is the total cost associated with those templates.

The Rule Base. The rule base is subdivided into three sets of rules. The first set is consulted before the application of an instruction in order to determine if the candidate instruction is applicable in the current context. In Section 4, we noted that such rules are necessary to distinguish among instructions with the same preconditions and postconditions. Rules are also useful to help reduce the effective branching factor. For instance, both instructions APAC (add contents
of register P to the contents of the accumulator) and ADD (add contents of a memory location to the contents of the accumulator) could be used to cover \((x + b \times c)\) in Example 3. APAC’s precondition requires that one operand be in register P and the other in the accumulator. However, the architecture of TMS320 does not provide an easy way for loading register P with a desired value \(d\). In the best case, \(d\) is in register T and one could use instruction MPYK_1 to multiply \(d\) by 1, with the result stored in register P. Hence, one rule specifies that APAC be considered only when one of the operands is already in either registers P or T. Otherwise, ADD is certain to be less expensive and APAC is pruned from the search.

The second set of rules is consulted after the application of an instruction, to determine if additional actions should be taken. For example, consider instruction MAC_pma_dma. This instruction first adds the contents of register P with the contents of the accumulator, leaving the sum in the accumulator; then, it replaces the contents of register P with the product of two operands, one addressed by PMA and the other by DMA. Now consider the two statements in Example 1. After covering statement S1 with MPY, the product \((b \times c)\) is left in register P and the new value of \(x\) is left in the accumulator. In statement S2, after covering \((d \times e)\) with MAC, one of the postaction rules matches, because the portion \((x + b \times c)\) can be readily covered by MAC. Hence, the action part of that rule will also mark node 11 covered, as a postaction to marking node 13 covered.

The third set of rules is used during algorithm transformations and will be discussed in the next section.

4.2 Time-Complexity Analysis

Although we stated in Section 1 that compilation speed should not be a major design constraint, we must not overlook that the entire algorithm must be executable within practical time limits. Therefore, we must analyze the time complexity of the proposed algorithm.

**Theorem 1** The evaluation function defined by expression (1) executes in time linearly proportional to the DAG size \(S\) and exponentially proportional to the look-ahead factor \(K\).

**Proof.** Expression 4 involves a count of the number of templates and their associated costs. It is independent of the DAG size or the look-ahead factor. Assuming that a node expands, in the worst case, into \(B\) successor nodes, expression (2) involves \(O(B^6)\) calls to expression (4) and hence it will execute in \(O(B^6)\)
steps. In expression (1), $F(N)$ involves a DAG traversal to count the number of operands and hence will execute in $O(S)$; $\text{NC}_2(N)$ is computed by expressions (2) and (3), and thus will execute in $O(B^k)$. Analogous to (4), the computation of $I(N)$ and $G(N)$ is independent of $S$ and $K$. Consequently, expression (1) executes in $O(S + B^k)$ steps.

**Theorem 2** The entire search algorithm can be accomplished in polynomial time with the DAG size $S$ and exponentially proportional with the look-ahead factor $K$.

**Proof.** The worst case is when each template covers only one DAG node. From the search tree in Figure 11a, if a node expands (in the worst case) into $B$ successors and the time necessary to evaluate the cost of a successor node is $T$, one can readily derive that the (worst case) total search time is given by

$$\tau = \text{total search time} = S \text{ (time to expand one node)} = S(BT)$$

(5)

Because $T$ is proportional to $O(S + B^k)$ (from Theorem 1), the total search time is proportional to $O(BS^2 + B^{k+1}S)$.

Hence, for reasonably small (fixed) values of $K$, the proposed algorithm executes in polynomial time with the DAG size.

### 4.3 Summary and Conclusions

In this section, we have presented in detail a methodology for code generation for PDSPs. Our approach combines traditional code generation methods, namely tree/DAG traversal and template pattern matching, with artificial intelligence techniques, namely MEA, HP, expert system, and heuristic search, to mimic the approach used by human PDSP assembly programmers. A time–cost analysis ensured that the proposed algorithm is bounded and feasible within practical limits of time. OASIS executes in time proportional to a second-order polynomial on the program size.

So far, we have introduced an efficient approach for code generation. By combining the subproblems of scheduling, instruction selection, and resource allocation into a single heuristic search framework, our approach can potentially generate optimal code for a given DAG. The need for post-code-generation optimization is obviated by the heuristic search formulation. Better yet, important

---

* A $K$-level tree rooted at one node, with each node expanding into $B$ successors, has exactly $(B^k + B^{k-1} + \ldots + B + 1)$ nodes. For simplicity, it is common practice to approximate the total number of nodes to $O(B^k)$. This approximation becomes more accurate as $K$ increases.
optimization issues that conventional optimizing compilers must deal with, such as how to ensure that an optimization technique will not undo the results of a previous optimization, are absent.

However, handwritten quality code generation still hinges on one assumption—that the input HLL program is a good representation of the intended DSP algorithm. In other words, the generated code can only be as good as the input code allows it to be.

To alleviate this constraint, we propose the use of algorithm transformations. In the next section, we go one step further and introduce the idea of context-sensitive algorithm transformations, examine some complicating issues involved, and show how it can be easily incorporated into our methodology.

5 CONTEXT-DEPENDENT ALGORITHM TRANSFORMATIONS

Even if a code generator could generate optimal code for a given HLL program, that code may still be inferior to handwritten assembly code. The reason is simple. Although the IR can uniquely represent a given sequence of HLL instructions (program), the latter is not a unique implementation of a desired algorithm. In fact, there are infinite programs that evaluate a given expression. For example, one could implement \((a \times b)\) as \([((a + b)^2 - a^2 - b^2)/2\). Incidentally, Massalin’s Superoptimizer [71] can generate optimal instruction sequences given a function to be performed. Superoptimizer, however, is not a code generator. It takes as input a machine language program and returns another (smaller) program, in the same machine language, which computes the same function as the input. It does not understand the intended function. It simply performs an exhaustive search over the target machine’s instruction set for all possible equivalent programs, first of length 1, then of length 2, and so on. For each function, the user must define a set of test vectors to be used to verify program equivalence and manually inspect the programs that pass such a test for equivalence with the original program. Due to its exponential nature, its application is limited to very small programs. Massalin wrote: ‘‘The current version of Superoptimizer has generated programs 12 instructions long in several hours running time. . . . Therefore, the Superoptimizer has limited usefulness as a code generator for a compiler.’’

The use of HLLs emphasizes the need for algorithm transformation. Unlike assembly languages, HLLs are, in principle, processor independent. Without target architecture information, HLL programmers cannot bias the program toward certain constructs, as experienced assembly programmers often do. Hence, it is the responsibility of the compiler to perform that task.

The term algorithm transformation has a broad meaning. Research on
transformations can be traced in many related areas, such as numerical analysis and algebra [72,73], high-level synthesis [74,75], CAD design [76,77], and software compilers [64,78–80]. We will obviously focus our attention on transformations in software compilers, also known by another equally broad term—optimizations. In this section, we will discuss only context-dependent transformations. See Section 2 for a survey of optimizations and their classifications in a more general sense.

In static transformations, the compiler uses static analysis to determine the merits of a transformation. For instance, the apparently more complex implementation \( \frac{(a + b)^2 - a^2 - b^2}{2} \) for \( (a \times b) \) may actually make sense if the target processor is an analog computer. In such a machine, there may not be direct hardware support for multiplication. On the other hand, addition and division can be easily implemented with resistors and operational amplifiers, and the exponential characteristics of certain analog elements can be exploited to implement the squaring operation. In such an environment, the transformation \( (a \times b) \rightarrow \frac{(a + b)^2 - a^2 - b^2}{2} \) is context independent, and as such, the compiler could simply replace, at parse time, all instances of \((a \times b)\) with \((a + b)^2 - a^2 - b^2)/2\).

In contrast, context-dependent transformations are those which depend on either the values of the operands or the contents and status of the available resources such as registers and memory addresses at run time. Such transformations are usually undecidable at parse time. For example, consider a processor in which subtraction and right shift take one machine cycle each, whereas multiplication takes four machine cycles. In most circumstances, implementing \( (a \times b) \) as \( \frac{(a + b)^2 - a^2 - b^2}{2} \) would not make sense in such a machine. However, if the partial results \( (a + b)^2, a^2, \) and \( b^2 \) have been evaluated in previous statements and are still available in appropriate registers, the code for \( (a + b)^2 - a^2 - b^2)/2 \) could execute in three machine cycles as opposed to four machine cycles for \( (a \times b) \).

In the framework for code generation presented in Sections 3 and 4, the HLL program is first parsed into a DAG. Our system makes use of artificial intelligence techniques (means-ends analysis, hierarchical planning, expert system, and heuristic search) to mimic the reasoning and planning processes used by human assembly programmers. Similar to template pattern matching (TPM) (see Sect. 2.1.2), all evaluations are performed at compile time. Unlike TPM, however, template costs are dynamically obtained. This means that a template may have different costs for different program contexts. Nevertheless, the ability of human assembly programmers to understand the algorithm being coded and modify certain parts of it to suit the target architecture was not supported. We now present an extension to our methodology for supporting algorithm transformations and describe how it can be easily integrated into our system.
5.1 Algorithm Transformations

Algorithm transformation in software compilation is the process of rewriting a given implementation of an algorithm (or parts of it) into another implementation that allows the generation of more efficient code. For our purposes, efficient means compact and fast. Twaddell [81] writes “the long-term trend in optimizations is for the compiler to effectively rewrite the code however it pleases with the user’s code representing just and expression of intent.”

Algorithm transformations can be classified into four major groups, as shown in Table 1. In this subsection, we present a framework that can support algebraic and Boolean transformations. In our approach for code generation, register allocation, instruction selection, and scheduling (of IR operations) are handled concurrently. The context-dependent search formulation assures that the sequence of instructions generated is optimized in its ordering. Hence, the value of a separate execution reordering phase is much reduced. However, that is not to say that separate instruction reordering is unnecessary. It is possible that executing certain instructions out of their original order, as long as the new order does not disturb the content of particular resources, can reduce pipeline stalls in some architectures. The need for instruction scheduling is further reduced by modern out-of-order-execution design technology, in which the processor is capable of executing any given sequence of instructions in the order that it considers best, without affecting the correctness of the result [82,83].

5.1.1 Context-Dependent Transformations

Some of the transformations in Table 1 are performed by conventional optimizing compilers. In these compilers, however, transformations are static and context

<table>
<thead>
<tr>
<th>Classes</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algebraic transformations</td>
<td>• Associativity, commutativity</td>
</tr>
<tr>
<td></td>
<td>• Identity element, neutral element</td>
</tr>
<tr>
<td></td>
<td>• Strength reduction, constant folding</td>
</tr>
<tr>
<td></td>
<td>Example: (a + (b - c) = b - (c - a))</td>
</tr>
<tr>
<td>Boolean transformations</td>
<td>• Associativity, commutativity</td>
</tr>
<tr>
<td></td>
<td>• DeMorgan’s Law</td>
</tr>
<tr>
<td></td>
<td>Example: (A \cdot B = A + B)</td>
</tr>
<tr>
<td>Execution reordering</td>
<td>• Instruction scheduling</td>
</tr>
<tr>
<td>Loop restructuring</td>
<td>• Loop fission, loop fusion,</td>
</tr>
<tr>
<td></td>
<td>• Unimodulo transformations</td>
</tr>
<tr>
<td></td>
<td>• Loop unfolding, retiming</td>
</tr>
</tbody>
</table>
insensitive. For example, if multiplication executes much slower than addition in the target processor, subtree \((a \times 2)\) could be replaced by subtree \((a + a)\). This transformation can be done at parse time and is context independent. However, if both operations perform in the same number of cycles, the advantage of such transformation is unrecognizable at parse time.

A common approach in these compilers is to parse the IR before code is generated and perform transformations wherever possible, assuming that such transformations will always benefit code generation. If that assumption proves to be incorrect, the user must disable specific transformations and recompile. Such an approach has two shortcomings. First, determining which transformations are responsible for the resulting inefficiency of the code is not trivial. Second, it cannot address the issue of a given transformation being advantageous in certain parts of the program but undesirable in others. We call the latter context-dependent transformations.

Another difficulty with context-dependent transformations is that the code generator must decide not only if a certain transformation is worth performing but also, in case several alternative transformations exist, which one to perform.

Our heuristic search framework can be easily extended to support this issue. Currently, each search node contains a copy of the remaining DAG to be covered. A pointer called \texttt{CurrentNode} points to the node in that DAG currently being covered. If \(N\) alternative patterns exist for the node pointed to by \texttt{CurrentNode}, the search node is replicated \(N\) times. Each replica contains a copy of the DAG, with the sub-DAG pointed to by \texttt{CurrentNode} replaced by one of the alternative patterns. A look-ahead heuristic search is then conducted for each replica and the least expensive among all replicas is preserved and others are discarded.

### 5.1.2 Reducing Search

Figure 12 lists the transformations currently implemented in OASIS. The column Condition in Figure 12 conveys heuristics to minimize the search space. A transformation is considered only if the listed condition is satisfied. If the condition is not satisfied, the transformation most likely yields either less compact or a slower code, and hence is ignored.

For example, in transformation 6, there are two alternatives: \(\text{Op1} - (\text{Op2 + Op3}) \rightarrow (\text{Op1} - \text{Op2}) - \text{Op3}\) and \(\text{Op1} - (\text{Op2 + Op3}) \rightarrow (\text{Op1} - \text{Op3}) - \text{Op2}\). First, before these transformations can even be considered, the required condition that the accumulator contains Op1 must be satisfied. Second, if the condition is satisfied, which one of the two alternative transformations will yield better code depends on the run-time context. Both alternatives are evaluated (with \(K\)-step look-ahead) together with the original expression and the least expensive among the three implementations is retained and the others are discarded.

In another example, transformation 7 also has two possible alternatives: \(\text{Op1} - (\text{Op2 - Op3}) \rightarrow (\text{Op1} - \text{Op2}) + \text{Op3}\) and \(\text{Op1} - (\text{Op2 - Op3}) \rightarrow\)
<table>
<thead>
<tr>
<th>Transf.</th>
<th>Original Pattern</th>
<th>Transformed Pattern</th>
<th>Condition</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Op1 Op2</td>
<td>Op2 Op1</td>
<td>a(\text{Op}_1 = 0), b(\text{Op}_2 = 0)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c\text{RegP} - \text{Op1} II \text{ACC} = \text{Op2}</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>d(\text{Op1} = \text{Op2})</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Op1 Op2</td>
<td>Op1 Op2</td>
<td>e(\text{NegHas} = 1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>f(\text{Op1} = 2\times \text{Op2})</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Op1 Op2</td>
<td>Op1 Op2</td>
<td>g(\text{Op1} = 2\times \text{Op2})</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>h(\text{ACC} = \text{Op2})</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Op1 Op2 Op3</td>
<td>Op1 Op2 Op3</td>
<td>i(\text{ACC} = \text{Op1})</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>j(\text{ACC} = \text{Op3})</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Op1 Op2 Op3</td>
<td>Op1 Op2 Op3</td>
<td>k(\text{ACC} = \text{Op1})</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>l(\text{ACC} = \text{Op3})</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Op1 Op2 Op3</td>
<td>Op1 Op2 Op3</td>
<td>m(\text{ACC} = \text{Op1})</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n(\text{ACC} = \text{Op3})</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>x = A + B</td>
<td>x = A + B</td>
<td>o(\text{ACC} = x)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>y = A + B</td>
<td>y = x + 2B</td>
<td>p(\text{ACC} = y)</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 12**  Algorithm transformations performed by OASIS during instruction selection.
(Op3 − Op2) + Op1. The content of the accumulator dictates which alternative (if any) should be used. The search process then determines if the original or the transformed expression will yield better code.

Transformations are classified into two levels. A level-1 transformation is one that can be implemented without modifying the DAG, whereas a level-2 transformation involves DAG modification. For example, transformation 2, Op1 − Op2 → Op1 + (−Op2), can be accomplished by adding such templates as ADD_dma and APAC into the connection table associated with the subtraction operator, under the condition that Op2’s **Negative** flag is set (by a previous operation). No modification in the DAG is necessary. However, the alternative Op1 − Op2 → − (Op2 − Op1) involves modification of the original DAG and is therefore classified as level 2. The main reason for classifying transformations according to changes in the DAG is to give the user more flexibility in matching computational resources. Level-1 transformations are computationally much less demanding than level-2 transformations. The former adds a new branch in the search tree, whereas the latter requires replicating the entire search tree.

5.1.3 Delayed Common Subexpression Elimination

Common subexpression elimination (CSE) is a popular technique applied by many optimizing compilers. In CSE, the compiler first determines the existence of subexpressions that are common to several parts of the program. If such subexpressions exist, the representation of the program is altered such that only the first occurrence of that subexpression is evaluated. The result is saved as a temporary value to be used by all other occurrences, eliminating multiple recomputations of a same expression.

In performing context-dependent optimizations, we must deal with three important issues. First, applying transformations during code generation can lead to what we call delayed CSE. For example, consider the code fragment in Example 4.

**Example 4**

\[
S1: \quad f = a + (b - c); \\
S2: \quad g = a - (b + c);
\]

Static CSE would not detect any common subexpressions. However, if we apply transformations 5 and 6 from Figure 12, the above code would result in

\[
S1': \quad f = b + (a - c); \\
S2': \quad g = (a - c) - b;
\]
and the common subexpression \((a - c)\) is created. In our code generation formulation, performing delayed CSE is a simple matter of traversing the DAG after each transformation is applied and merging common subexpressions as they are encountered. This leads to the second issue.

Depending on the subexpression, the resource status and the target architecture, recomputing the expression could be less expensive than storing and reloading a value from memory. The correct decision, to perform CSE or not, can only be made after both alternatives are evaluated. Our code generation formulation supports such evaluations through heuristic search.

The third issue is that of look-ahead, which must be used to further refine the evaluation. Let us consider Example 5.

**Example 5**

\[
\begin{align*}
S_1: & \quad f = b + (a - c); \\
S_2: & \quad g = a - (b + c);
\end{align*}
\]

Suppose we are processing statement \(S_1\) and that transformation 5 is applicable (i.e., the accumulator holds the value of variable \(b\)). Indeed, transforming \(S_1\) into

\[
S_1': \quad f = a + (b - c);
\]

is advantageous because it eliminates the need to store \(b\) into a temporary location and reloading the accumulator with the value of \(a\) to compute \((a - c)\). A naive evaluation function would opt to transform \(S_1\). However, only by looking ahead can one see that if transformation 5 is not applied and transformation 6 is applied on \(S_2\) instead, the result would be

\[
\begin{align*}
S_1: & \quad f = b + (a - c); \\
S_2': & \quad g = (a - c) - b;
\end{align*}
\]

and performing the delayed CSE on \((a - c)\) could be potentially more advantageous, especially if \(a\) and \(c\) are complicated subexpressions.

OASIS supports heuristic search with user specified look-ahead levels.

**5.2. Time-Complexity Analysis**

With the implementation of algorithm transformations as replication of search trees, it would seem that context-sensitive transformations could be too expensive in practice. In fact, one could wonder if the algorithm now is still bounded. In
other words, is the new algorithm still realizable within feasible amounts of time? To ensure the feasibility of the algorithm, we conduct the following time–cost analysis

**Theorem 3** With CDAT, the proposed algorithm still executes in polynomial time with the DAG size $S$ and time exponentially proportional with the look-ahead factor $K$.

Proof. In the worst case, every search node has $Z$ alternative transformations and each transformation requires replication of the entire branch rooted at that search node. Following the rationale in the proof of Theorem 2, the time necessary to expand one node is multiplied by $Z$, because each node now can have $Z$ replicas. Similarly, the time necessary to evaluate the cost of a successor node, $T$ is also multiplied by $Z$. Hence,

$$\tau = \text{total search time with transformations}$$

$$= SZ[B(ZT)] \approx O(Z^{2}[BS^2 + B^{K+1}S])$$

(6)

5.3 Summary and Conclusions

In this subsection, we presented an extension to the heuristic search framework for code generation presented in Sections 3 and 4. This extension allows the code generator to consider context-dependent algebraic and Boolean algorithm transformations. The implementation is straightforward, based on recursive calls to the same search engine introduced in Section 4.

The only drawback is that more look-ahead levels are required when transformations are involved. Although our heuristic search formulation executes in (second order) polynomial time with the program size, it is exponential with the look-ahead factor. For time-critical applications, transformations are well worth the increased compilation time.

In the next section, we present some empirical results obtained with OASIS, a prototype code generator based on the ideas described in this and the previous two sections. The results will demonstrate quantitatively the efficacy of our methodology.

6 EXPERIMENTAL RESULTS

A prototype code generator, OASIS, following the ideas discussed in Sections 3–5, has been implemented. We targeted it to a simplified architecture, based on the Texas Instruments TMS3202x/5x [84]. The primary purpose of OASIS is to
evaluate the feasibility of the proposed approach. It is not intended to be a fully working compiler.

6.1 Quality of the Generated Code

Two sets of benchmarks involving common DSP algorithms have been designed to evaluate the performance of our code generation approach. Whenever possible, we selected published DSP routines that have been coded in assembly by DSP experts. In the absence of published material, we coded the routines in assembly, with a single purpose in mind: to produce the most efficient code possible. For that purpose, we utilized all possible transformations that we could apply. This set of handwritten assembly codes serves as our benchmarks.

The DSP algorithms implemented by those routines (not necessarily the assembly codes) were coded in C and supplied to both OASIS and the Texas Instruments TMS3202x/5x Optimizing C Compiler [85].

Table 2 contains the results of the first set of benchmarks. The codes gener-
ated by OASIS were obtained without algorithm transformations. The TI compiler ran with all optimizations enabled.

The code generated by OASIS is up to 3.8 times smaller (in benchmark BIQUAD) than that obtained from the TI compiler. Furthermore, OASIS is capable of generating code whose quality is comparable, in some cases even superior, to that generated by human DSP experts.

Note that in the benchmark ROTATION, although OASIS generated a few more instructions, the program executes faster than the handwritten version. This is, in part, due to our heuristic function, which minimizes the sum of code size and execution time.

Except for FFT2, the other six benchmarks used in Table 2 do not lend themselves well to testing CDAT (discussed in Sec. 5). Hence, a new set of benchmarks was devised, specifically to test the efficacy of the proposed context-dependent transformations and the issues involved when delayed CSE is present. Table 3 summarizes the results. Along with the size of the generated code, it also lists the smallest look-ahead factor \( K \) necessary to obtain those results.

Without transformations, the code generated by OASIS is already much smaller than that obtained from the TI compiler, but not quite as small as the handwritten version. The reason is that when coding in assembly, the programmer has access and, indeed, takes advantage of all kinds of "tricks" (algorithm trans-

---

**Table 3** Performance of OASIS Compared with TI Compiler and Handwritten Assembly Code

<table>
<thead>
<tr>
<th>HLL program</th>
<th>TI compiler(^a) (# instr.)</th>
<th>Without transf. (# instr.)</th>
<th>With transf. (# instr.)</th>
<th>Handwritten assembly (# instr.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xform1</td>
<td>18</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>xform2</td>
<td>18</td>
<td>9</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>xform3</td>
<td>27</td>
<td>14</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>xform4</td>
<td>24</td>
<td>9</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>xform5</td>
<td>23</td>
<td>10</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>xform6</td>
<td>27</td>
<td>14</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>FFT2</td>
<td>63</td>
<td>23</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>FFT4</td>
<td>190</td>
<td>107</td>
<td>3</td>
<td>89</td>
</tr>
</tbody>
</table>

\(^a\) The TI compiler is capable of static arithmetic transformations.
\(^b\) Results obtained from Ref. 86.
\(^c\) Results obtained from Ref. 91.

Note: The number of instructions excludes assembler directives and routine initialization instructions. Results obtained with all optimizations enabled (level 2).
formations) that one can envision. However, with context-dependent transformations enabled, the prototype generates codes that are truly comparable to their handwritten counterparts. FFT4 is an exception. Due to its size (over 100 nodes in its DAG), we exceeded the available computational resources before we reached the limitations of our code generator. Analysis of the generated code shows that not all possible optimization opportunities have been performed by OASIS. On the other hand, OASIS generated smaller codes for xform3 and FFT2 than the assembly programmers.

Careful examination of the generated assembly code for xform3 reveals that OASIS took advantage of delayed common subexpression elimination and the programmer missed that optimization opportunity in the handwritten version. This example illustrates well the problems faced by programmers. Hand coding in assembly language gives the programmer tremendous flexibility in performing algorithm transformations. Although human ingenuity is far superior than what current technology can give to a computer program, the latter can perform certain tasks much faster, more reliably, and more thoroughly than we can. An automated and consistent approach can uncover optimizations that are hidden deep within the program.

### 6.2 Summary and Conclusions

The results presented in this section show that our code generator is able to generate code that is as good as that generated by human DSP experts. Our approach allows a great degree of flexibility in matching available computational resources to the desired output code quality. By varying the weights of individual features in the heuristic function, one can generate code that executes fastest (e.g., for real-time applications) or which has the smallest size (e.g., to fit into a limited size PROM), or both. By increasing the depth of the search, our approach is capable of yielding globally optimized code, albeit in time exponentially proportional to the search depth. From the previous results, without algorithm transformations, a search depth of 3 is adequate for most cases and compiles in time polynomial in the input program size.

The experimental results also confirm the efficacy of context-dependent algorithm transformations. The only drawback is that more look-ahead levels may be required when transformations are involved. Although our heuristic search formulation executes in (second-order) polynomial time with the program size, it is exponential with the look-ahead factor. For time-critical applications, transformations are well worth the increased computational time.

Previous approaches are restricted in time and scope. An exhaustive code generation approach may require more computational resources than currently available [31,71]. A fast approach developed for today’s processing power cannot
generate the desired quality [7,8]. Worse yet, it may be very quickly outdated given the pace at which computational power is advancing.

Although no specific limits exist on input program size, this model is especially suited for tasks characterized by small routines with very stringent constraints on execution time and/or code size.

Advantages of the proposed approach include the following:

1. No need for postgeneration optimizations. Unlike conventional compilers, instruction selection and resource allocation are combined into one single pass, assisting and influencing each other. Many optimization techniques implemented in conventional optimizing compilers are trivialized by the search formulation. A few optimizations are implemented in the form of CDAT. Unlike static optimizations, CDAT can guarantee to improve code quality. The problem of applying optimizations in the correct order so that one optimization does not undo the results of previous optimizations is also trivialized.
2. Potential for obtaining optimal code. The search strategy is flexible, in the sense that it allows a trade-off between code quality and compilation time.
3. Enhanced portability. Retargeting the code generator to different PDSPs requires modifying only two modules: the instruction library and the rule base.

REFERENCES

61. CW Fraser, DR Hanson. A code generation interface for ANSI C, Department of Computer Science, Princeton University, Research report CS-TR-270-90, 1990.
89. C Crowell. Matrix multiplication with the TMS32010 and TMS32020. In: K-S Lin,