Abstract

A new hardware prototyper is described for rapid retargeting of hardware and compilers. Architects can directly make machine organization via compiler support design trade-offs. This novel rapid prototyping tool features relatively simple hosting (on a PC-AT), target architectures as diverse as data flow machines, and minimal use of host memory. This new tool emphasizes simple and rapid retargeting through careful exploitation of code transformations and expansions internal to the tool. An architect would use this tool to test new computing hardware by directly simulating the applications code written in C. Optimization techniques are applied both at the language-dependent level to the intermediate form by shape analysis and during code generation through cost analysis. A major advantage of the tool is the significant reduction in microcode development time, which translates directly into lower economic costs.

I. Exploitable AI Within Rapid Prototypers

Rapid prototyping systems still largely lack an integrated design basis. Space Tech Corporation has been researching non-Von architectures for several years. From this effort, we (as a 20 year architecture house) have seen many unconventional yet extremely powerful architectures fail to achieve notoriety because software design tools were lacking. It is timely to offer a rapid architectural prototype tool to architects for enhancing systolic array, vector, parallel, connectionist, hypercube, etc. machine that allows compiler issues to influence the hardware design process. Most tools are either hardware or software driven, yet except for the RISC architecture virtually no current general-purpose architectures have been designed concurrently with true hardware/software integration in the public domain. Furthermore, today’s CAD and CASE VLSI design tools are predominantly fragmented basing their performance upon either behavioral, structural, or physical requirements. Our approach is to make a low-cost rapid prototyping VLSI design tool that exploits the best features in AI within the design tool inference engines. Because compiler generation is a significant cost, our solution integrates both software programming specifications with hardware performance specifications. We are motivated by an automated tool for program specification and hardware design. We believe that judicious employment of AI internal to the tool is warranted. Briefly, our innovation is the merge of expert knowledge/rule-based inference engines, flow analyzer, code generator, and compactor of a rapid prototyper so that eventual compiler generation is less costly and the resultant hardware enhances parallel languages.

Software engineers recognize that the software design problem is, in fact, a system problem. To remain competitive, the United States must develop such rapid prototyping tools for our cottage industries in small and medium-size software development organizations. The Software Productivity Consortium in Europe and MCC recognize these opportunities. Likewise, the Eastport study group identified the gap in systems architecture and battle management software highly recommending that further development proceed in parallel so that hardware is designed to be best suited to software power. Software engineers must now go beyond compilers, debuggers, and utilities for source code control to help automate latter phases of programming. Johnson, Vice President at DEC, states that “Artificial Intelligence is simply the next logical step in data processing”.

AI can: (1) capture and retain expertise, (2) amplify expertise needed to successfully deploy new technologies and design applications, (3) offer systems that reason intelligently about necessary actions. The AI applications base unfortunately requires a strong base of development tools which have only recently appeared (e.g., KEE, Design Advisor, and SYNAPSE). AI benefits include explicit representation of symbolic structures, integration of various types of representation, visibility, reasoning in radically different ways from those attempted by non-AI tool systems, intelligent interaction based on representation and flexibility from the exquisite declarative nature of the representation.

The fundamental first step is the organization of facts into a logical format (taxonomy). The next step is to elicit from the taxonomy the facts and inferences needed to reach a hardware design decision. Briefly, our approach is to develop a language driven tool which innovatively separates us from the current plethora of prototyping systems, such as silicon compilers,
Microcode

Even firmware engineering tool capability can be RAMS, RISC architectures, and new microprocessors, has little more than a redefinable assembler architecture tools are necessary in order for the MacPitts, Palladio, First, Design Consultant, Socrates, Design Compiler, Smart Models, etc. It migration; (2) have features which allow user a trend already well established.

machines to be explored includes simple arithmetic non-standard architectural features rapidly; (11. intermediate technological developments, such as high micro-operations for analysis of machine hardware (ASIC).

Our approach goes far beyond the "integrated" tool packages of WCH, Daisy, Mentor, and Trimmer. We have been motivated by the ongoing technological developments, such as high density CMOS, new static memory technology, megabit EAMs, RISC architectures, and new microprocessors, a trend already well established.

II. A Rapid Architectural Prototype (RAP)

Space Tech Corporation has been developing RAP for over two years. It is about to be brought to a proof-of-concept stage directed towards specific architectures with full capability of an expert system for the user. RAP is shown in Figure 1.

It has long been known that non-Von VLSI architecture tools are necessary in order for the design effort to be more effective [1]. Today, one has little more than a redefinable assembler or editor to aid in the development of microcode, e.g. Even firmware engineering tool capability can be increased by developing microprograms in a CAD environment and then translating an HLL program to microcode [2]. The major difficulty is in producing a system which is viable for diverse architectures. Additionally, inherent architecture concurrency handling needs to be sophisticated. These tools must: (1) accept a well-known HLL for algorithm specification to promote acceptance of the tool and to provide support for vertical migration; (2) have features which allow user extension of the language to facilitate unusual or non-standard architectural features rapidly; (3) be retargetable to a wide range of architectures. These should include general purpose horizontal micro-instruction machines with both mono-phase and poly-phase timing characteristic; (4) produce optimized symbolic and binary microcode for forward and reverse trace through application code; (5) report on AI reasoning applied to the selection of micro-operations for analysis of machine description correctness and appropriateness of a particular coding strategy; and (6) provide means for providing diagnostic information for debugging of parallel architectures. Much more work is required in compiler assisted dependence analysis than in developing debugging tools [3]. Our general-purpose RAP meets these goals. The tool captures current software technology from the field of re-targetable compilation, CAD, CASE, KEE, concurrency recognition, optimization, and microcode compilation for SIMD, MIMD, vector, and array processors. Space Tech's RAP can recognize a subset of the programming language, and can be made capable of generating object code for array processors. A reasonable subset for parallel machines to be explored includes simple arithmetic expressions, assignments and if-then-else control structures.

III. Intermediate Semantic Description Language (ISDL)

The most critical concept in the "intermediate language". Our ISDL is a special parse tree that is a directed graph which represents statement flow. For the ISDL to truly be intermediate, it must have the following characteristics: (1) the ISDL must make no assumptions about the source language constructs used (language-independent), and (2) the ISDL must make no assumptions about the target machine architecture (machine-independent).

Some benefits of creating an intermediate language with these characteristics are: (1) retargeting is facilitated. If a new language is desired, only the front-end need be redesigned. If a new target machine is desired, only the back-end need be redesigned. (2) A machine-independent optimizer can be applied to the intermediate code. Each parent node of the graph represents a control statement for execution flow, or a program operator, such as add, subtract, or assignment. Leaf nodes represent operands to each parent node, and may be parents of further nodes or identifiers and constants. The ISDL is a proof-of-concept at the source code is parsed, thus requiring no separate phase for intermediate code generation.

IV. Pre-Resource Allocation

This component prepares resource and variable information for the Code Generator resource allocation scheme. Input to this component is the modified ISDL code package from the pass through phase, resource tables, and data entry derived from the machine description. The pass-through phase has two functions. First, a linked list of free, general purpose storage elements is created. Second, the ISDL code is scanned, and variable information is initialized for each user variable in the code. A total frequency count for each variable is measured for use by the resource allocation algorithm. Additionally, if a main memory resource is present, each variable is assigned a location(s) in the main memory for deallocation purposes. The Pre-Resource Allocation is responsible for assuring that the variable has allocated sufficient words of memory to meet the data type requirements of the variable.

V. Resource Allocation

The RAP Resource Allocation scheme is adopted from [6]. Variables are classified according to five properties: 1. Status: the variable is active if the contents of its storage are the same as main memory; or passive if different; 2. Scope: if the variable is seen outside the current block, it is of global scope; otherwise, of local scope; 3. Type: the variable is busy if it has been referenced and will be referenced again inside the current block of code; otherwise, it is latent. 4. Explicit: a variable is explicit if it directly refers to a particular machine resource. This is a modification to the classification scheme in [1]. It was incorporated to allow explicit references to machine resources from the HLL program and to assure availability of explicit resources when requested; 5. Frequency: the number of times the variable has actually been seen is the frequency count. The Resource Allocation package is invoked whenever the Code Generator requests that a user of compiler-generated variable be allocated. The Resource Allocator determines whether or not the variable has been allocated. If not, it seeks to find free storage for the variable. If no free storage exists, it scans the currently allocated variables and uses the above attributes to determine which variable should be replaced.
Performing resource allocation "on the fly" during code generation reduces the possibility of conflict between the resource requirements of user variables and code generator temporary variables. This is made possible by maintaining the history of variables and resources during code generation, as well as reducing the chance that all resources are being simultaneously occupied during the generation of code generator temporaries. Variable allocation or deallocation may cause a read or write of main memory. If this occurs, code will be generated for the read or write and a data dependency link is added. The data dependency link is added between the code being generated and the time the allocation or deallocation was requested. This structure is known as a dependency link.

The Resource Allocator and the Code Generator are intimately connected. The reason these are so strongly associated is to prevent the Resource Allocator from consuming all of the resources for compiler-generated temporaries. By having both the Resource Allocator and the Code Generator operate in the same pass, this type of resource conflict will be prevented. The Resource Allocator will require, as an input, a tabular description of the resources available. This table is produced by a Machine Description Parser.

A major design decision for the RAP is the approach to retargetable code generation to rapidly assess VLSI design trade-offs. This component has, by far, the most significant AI impact on the other component. Syntactic analysis, through compaction, is affected by the selection of the code generation technique. This is due to the form of the intermediate code and the types of dependency information generated by this phase. Several code generation techniques have been studied and are classified in [7] as interpretive code generation, pattern-matched code generation, and table-driven code generation. Interpretive code generation produces virtual machine code which is expanded into a real target machine code during run time.

Difficulties arise in retargetability because of the inseparability of machine description and code generation algorithms. Pattern-matched code generation matches user program semantics to machine instruction semantics. Both heuristic search techniques and parsing techniques have been developed to perform the semantic pattern matches. Table-driven code generation uses techniques which are similar to pattern-matched code generation. The major difference is the separation of a table-driving code generator from a table-driven code generator. The Code Generator matches the machine instructions found in the tables to the actual semantic description derived from the user program.

Interpretive code generation techniques have been evaluated as the most difficult in terms of retargetability. Retargetable interpreter code generation requires significant knowledge of the interpreter algorithms due to the inseparability of the machine description and the interpreter code.

The degree of optimization which can be achieved is also severely limited because virtual machine code cannot always reflect the largest possible program segment which can be handled by a single instruction on the target machine. Interpretive code generation has been eliminated as a possible alternative for RAP. Pattern-matched code generation and table-driven code generation offer similar advantages in the degree of retargetability. Table-driven code generation offers an advantage in speed by separating code generation into a compiler-compile time phase and a compile time code generation phase. This time savings increases implementation complexity.

Evaluation of the value of heuristic search techniques versus parsing techniques is more difficult to judge. Heuristic search techniques are simpler to implement and machine descriptions are less complex. Parsing techniques offer a greater degree of guidance in register selection, storage binding, and some machine-dependent optimization. The cost of parsing techniques in machine description and implementation complexity are deemed far too high for the first practical development of RAP. A heuristic search pattern-matched scheme was selected for the RAP Code Generator. The Code Generator processes an ISDL program by attempting to closely match each program statement with a semantic description of a machine micro-operation. Mismatch of operands are handled by further decomposition or axiomatic transformations. One or more micro-operations are emitted as a result of statement matches and operand subtargeting.

As code is generated, data dependency links are created between micro-operations which have implicit or explicit data dependencies. Implicit data dependencies arise when a statement contains an output variable which is used as input to a proceeding statement. Explicit data dependencies occur between micro-operations generated by a statement and micro-operations generated during the subtargeting of operands. The data dependency graphs so created are used during compaction to prevent data dependency violations in horizontal micro-instructions. The Code Generator receives the machine description from a machine description file. The file specifies three types of machine dependent information: (1) Machine Resources, (2) Concurrency Conflict Classes, (3) Micro-operation Semantic Descriptions. The machine description is used primarily by the resource allocation routines for storage assignment. The conflict class description provides a method of measuring the cost of a micro-operation in terms of the probability of concurrency conflict with other micro-operations. The micro-operation semantic description specifies the semantic meaning of each micro-operation. The semantic description is specified in ISDL. Each micro-operation is specified by a symbolic name and micro-operation field for horizontal placement.

A second file is used by the Code Generator during the subtargeting of operands. It is also used during rearrangement of a statement if a micro-operation match cannot be found. This file is a list of axioms. The axioms specify equivalence relationships such as the rules of commutativity and associativity.
Application of the Code Generator to an ISDL program produces a tree of micro-operations with data dependency links (in a vertical format). The micro-operations are a machine-dependent realization of an ISDL program. This tree is passed to the compactor for conversion to a horizontal micro-instruction format.

VII. Flow Analyzer

This component performs a flow analysis on an ISDL code package. Flow finds common subexpressions and rearranges code sequences for more optimal intermediate code. Constant expressions within loops are moved outside of the loop if data dependencies permit and side effects are not present. Segments of code which have no entrance are eliminated. A modified ISDL code package is produced from this phase. AI makes great contributions herein.

VIII. Compactor and Binary Code Generator

The compactor, to be developed, utilizes a greedy heuristic strategy for the labeling problem. Forward look-ahead techniques in an AI expert/knowledge based mode searched tables of machine resources and microcode segments for candidate micro-operations. Timing and data dependencies form the labeling constraints. In short, an optimal graph coloring occurs. Chapter 6 of the text [8] by this PI has been used to perform compiler construction techniques along with AI inference engines.

The Binary Code Generator is, in essence, a meta-assembler tool. This assembler is driven by a symbolic definition file which defines the symbolic micro-operation names as bit patterns overlayed in a micro-instruction word. The assembler takes care of default bit patterns for a micro-instruction containing missing field. The output of the Binary Code Generator is a relocatable binary code file. This file is input to a relocating linker loader to produce executable object code.

IX. Machine Description Parser

The Machine Description Parser is a LALR(1) parser which parses the target machine description. The target machine description can be specified in Machine Description Language (MDL). MDL allows specification of a microprogrammable machine in three parts: the machine resources, concurrency conflict classes, and micro-operation semantics. The motivation for producing a Machine Description Parser is three-fold. First, reargetability of the Code Generator is desired. This is achieved by using a machine-independent Code Generator which accepts the source input (in this case the ISDL program), and a machine description table. Conceptually, it is easier to separate the Code Generator into two phases due to the complexity of interpreting the procedural machine description at compile time. Second, separation into a compile-time and a compiler-time Machine Description Parser produces a faster and more efficient Code Generator. Third, a Machine Description Parser can provide tables of machine properties which will aid the resource allocation and compaction. The Machine Description Language, as designed, will create tables necessary for assembly code generation. The grammar can be modified rather easily to create binary code for a machine. The grammar for the Machine Description Language is based on Catell's work found in [9].

X. Declarative Knowledge and Meta-Knowledge in RAP

Declarative knowledge belongs in the inference engine for the flow analyzer. It represents a first kind of knowledge, relating simple relationships such as data flow amongst multiple ALU's. Declarative knowledge often uses theorem proving techniques for reasoning through the domain principles especially if no surface heuristics are available. Recognize that theorem proving is inherently expensive computationally hence, programmers find shortcut rules as to minimize "deep reasoning". Rather than invoke a situation-action paradigm we will use reduction operators. One objective is to build optimal inference engines using the reduction operators in the declarative knowledge base for the flow analyzer and study its efficiencies.

Meta-knowledge, often more difficult to elicit from an expert, invokes meta-rules (use of tags or flags attached to data) and an exquisite control languages to harness meta-knowledge. Meta-knowledge is more appropriately invoked in the code generator and compactor. Meta-knowledge produces focused progress in task performance and may sometimes be domain (architecture) specific. This still remains an active AI research area. One objective herein is to invoke meta-knowledge and meta-rules for trying to obtain procedural efficiencies within the code generator. Central to all knowledge bases is a frame-based lattice of machine classes to aid designer choices.

XI. AI Flow Analyzer

The Flow Optimizer phase performs a data flow analysis of the ISDL code produced for the source program. Specifically, the optimizer will search for common sub-expressions, rearrange operation sequencing, and decide certain basic issues, such as where registers might be optimally used. The Flow Optimizer will give the most payoff for the least effort. We believe that inference engine operating on the parse-tree using AI PROBLEM REDUCTION (AND/ OR) is best. See Table 1 for details.

XII. AI Code Generation

The Code Generation phase consists of two interdependent program packages. The first package is the Code Generator itself; and the second is the Resource Allocator. The decision to strongly associate these program modules is based on the restrictions that the Resource Allocator places on AP resource use. It is possible to design the Resource Allocator as a separate pass before Code Generation. In this configuration, the Resource Allocator could consume all of the machine resources for user variables. This would prevent the Code Generator from obtaining resources for compiler-generated temporary. Hence, the code generation and resource allocation phases have been associated to prevent this type of resource conflict in the AP.

The Symbolic Code Generator will generate a symbolic code representation from the ISDL program. This symbolic code can take on several forms, all of which are vertical in nature and take no advantage of inherent architectural concurrencies. The symbolic representation may be an assembly code or actual binary code with unresolved forward references which can be handled in a later phase. Algorithmically, the Code Generator operates as
follows:

Traverse ISDL program tree: (1) Match program tree node against the proper machine-dependent template, and (2) Take action specified by template matched.

This algorithm will traverse a node of the ISDL program tree looking for a template match. It will evaluate the degree of the match that took place (e.g., exact, close enough for subtargeting, etc.). It will then subtarget by calling itself on subtrees of the current tree node if exact matching was not found and attempt to find a match or perform a subtargeting on the subtree. In this way, the code generator will look for the largest possible ISDL. The template is a two-part structure with an LHS pattern tree and an RHS action sequence. The pattern tree is basically an operator root node with operand or operand closure leaf nodes. These nodes symbolically represent operations and source/sink combinations available on the machine. They may also represent decomposition functions such as reduction if-then-else blocks to a more machine-dependent level. The templates are derived from a machine description table and themselves are organized into a table for easy searching. The operator and operands are used to match against the operator and operands of the ISDL program tree. When an ISDL template match is found the action sequence is executed. The action sequence may specify code to be generated or compiler actions to be taken such as labeling a location in code.

In order to reduce problem search space, heuristic techniques, as well as other AI system techniques, will be applied to the code generator search algorithm. This is necessary because the tree search space is combinatorially large. Programs of even intermediate size and machines of moderate complexity could cost more in compile time than is gained in code efficiency. It can be shown that AI techniques can produce satisfactory and cost effective code when compared to the alternative compile time costs.

XIII. Reduction Operators (AND/OR)

Reduction operators are an important heuristic technique used in scene analysis. Haralick [10] and others have used these techniques to probabilistically solve NP-complete constraint satisfaction problems in polynomial time. The reduction operators can be used in a tree search of the problem space to find a solution that satisfies the required constraints of the problem. By expending a small amount of computational effort early in the search, fruitless solutions are discarded before an exhaustive enumeration is performed. Hence, very little backtracking occurs in the search for a solution. This is vital to the software compilation process. The primary advantage of using the reduction operators to solve the microprogram compilation problem is that the resulting algorithm is probabilistically optimum. Reduction operators work well in RAP. Mathematically, the problem is to discover all possible mappings of units into labels such that the assignments satisfy all of the constraints. This mapping is equivalent to a graph coloring, which is basically code optimization in software compilation.

Haralick and Shapiro [11, 10] discuss several definitions and mathematical concepts to describe the consistent labeling problem. Some of these concepts are important in the discussion of the techniques used to solve the consistent labeling problem for RAP. The problem consists of a set U of m units to be labeled. U = \{1, 2, ... , m\}. It also consists of a set of labels L. If \( u_1, u_2, ..., u_m \) are all members of the set with \( X \in L \) then the n-tuple \( (u_1, u_2, ..., u_n) \) is a labeling of the n-tuple of units \( (u_1, u_2, ..., u_n) \). Such a labeling is consistent labeling for \( n \) of the units if all of the resulting unit-label pairs are compatible. In other words, each label that is assigned to each unit in the labeling does not conflict with any other label assigned to its associated unit.

A globally consistent labeling is a consistent labeling for all \( n \) of the units to be labeled. To discover a globally consistent labeling, a compatibility model is used. A compatibility model aids in determining consistent labelings. This model indicates which units constrain one another \( N \) at a time where \( N \) is the degree of the constraints. It also indicates which labelings are permitted for these units which constrain one another \( N \) at a time. Often, much information is available that allows the compatibility model to be quickly developed.

Backtracking is a common method used to solve problems where a search must be made of many possible solutions to find the correct one subject to certain constraints. Backtracking uses a tree structure to represent the possible alternatives that need to be systematically searched. The nodes
of the tree represent entries to be added to a potential solution of the problem.

The consistent labeling problem can be modeled as a tree structure. The nodes each represent the assignment of a label to a unit. The path from the root node to another node represents the unit-label matchings made so far that are consistent with the one node. A consistent labeling of all the units to be labeled is the solution discovered in the tree search algorithm. One of the problems that backtracking algorithms are prone to is thrashing. Thrashing is the continued checking of a node at level j in the tree search against a previously established node and some level earlier than j despite the fact that the two nodes are known to conflict when the earlier node is assigned. It would reduce execution time of the search algorithm if the search could eliminate node j from consideration when the earlier node was assigned. Another problem that backtracking algorithms are prone to is excessive backtracking. If excessive backtracking occurs in a search to a solution, then the resulting algorithm complexity becomes exponential in the input size of the problem. We have devised some heuristic techniques known as reduction operators which will help eliminate thrashing and backtracking. These techniques look ahead and eliminate future assignments that are inconsistent with present assignments. The heuristic techniques that were developed are known as reduction operators. These heuristics behave in certain ways. These mechanisms are as follows: (1) Look-ahead and anticipate the future in order to succeed in the present, (2) Try to succeed, try first where you are most likely to fail, (3) Remember what you have done to avoid repeating the same mistake, and (4) Look ahead to the future in order not to worry about the past.

To understand how reduction operators work, it is first necessary to see how the \((U,L,T,R)\) model is used in searching for a solution without using reduction operators. First, the constraints and conditions of the problem are encoded into the \((U,L,T,R)\) model. The model provides all of the information necessary to search for a solution since all the problem constraints are present. Then, a unit is chosen from the set \(U\) to be labeled. Next, a label (e.g., \(L_j\)) is selected from the set \(L\) to assign to the unit. Now, the \(R\) relation of the model must be updated to reflect the assignment just made. By making the assignment, any other possible assignments of labels (e.g., \(L_1, L_2, \ldots\)) are no longer possible since only one label can be assigned to a unit. Therefore, the \(R\) relation must be restricted by eliminating all of the other possible choices since they no longer apply. This is accomplished by searching through all the entries in \(R\) and eliminating any entry that has this unit paired with a label that isn't the one selected. Note that the selection of the units and the labels can be done according to a priority scheme that optimizes the overall search time. This priority scheme is problem dependent.

Once the \(R\) relation has been restricted, we then choose an available unit from \(U\). Then, a label is chosen for the second unit. Now, it must be determined if the second unit-label pair is consistent with the first unit-label pair. This information is kept in the \(R\) relation. If an entry can be found in \(R\) that has the first unit-label pair associated with the second unit-label pair, then the assignment just made is consistent. Otherwise, the assignment made is not consistent. If the assignment made is not consistent, then another label is chosen for the second unit. If it is consistent, then another unit is chosen in the process repeated. If all the labels for a given unit are exhausted, then the search must backtrack to the first unit-label assignment and try a different label for the unit. This entails unrestricting the \(R\) relation and restricting it to another label for the first unit. The approach just described is a backtracking solution with the constraints of the problem solution kept available in the \(R\) relation. The reduction operators are used to limit the potential choices for future unassigned labels based on the present assignments.

There are two reduction operators available. These are the PHI and PSI operators works by taking entries from \(R\) one at a time and making sure that each entry satisfies certain conditions. If the conditions are not satisfied, then the entry is discarded. There are two parameters supplied with the reduction operators. These parameters are \(k\) and \(p\). The \(k\) parameter specifies the number of free units that must be checked in the test. This is an important proof in the current \(R\) relation. The \(p\) parameter specifies how many sets of free units taken with \(k\) units that must be checked in the test. This proof becomes exponential in the input size of the problem. We have developed some heuristic techniques known as reduction operators which will help eliminate thrashing and backtracking. These techniques look ahead and eliminate future assignments that are inconsistent with present assignments. The heuristic techniques that were developed are known as reduction operators. These heuristics behave in certain ways. These mechanisms are as follows: (1) Look-ahead and anticipate the future in order to succeed in the present, (2) Try to succeed, try first where you are most likely to fail, (3) Remember what you have done to avoid repeating the same mistake, and (4) Look ahead to the future in order not to worry about the past.

To understand how reduction operators work, it is first necessary to see how the \((U,L,T,R)\) model is used in searching for a solution without using reduction operators. First, the constraints and conditions of the problem are encoded into the \((U,L,T,R)\) model. The model provides all of the information necessary to search for a solution since all the problem constraints are present. Then, a unit is chosen from the set \(U\) to be labeled. Next, a label (e.g., \(L_j\)) is selected from the set \(L\) to assign to the unit. Now, the \(R\) relation of the model must be updated to reflect the assignment just made. By making the assignment, any other possible assignments of labels (e.g., \(L_1, L_2, \ldots\)) are no longer possible since only one label can be assigned to a unit. Therefore, the \(R\) relation must be restricted by eliminating all of the other possible choices since they no longer apply. This is accomplished by searching through all the entries in \(R\) and eliminating any entry that has this unit paired with a label that isn't the one selected. Note that the selection of the units and the labels can be done according to a priority scheme that optimizes the overall search time. This priority scheme is problem dependent.

Once the \(R\) relation has been restricted, we then choose an available unit from \(U\). Then, a label is chosen for the second unit. Now, it must be determined if the second unit-label pair is consistent with the first unit-label pair. This information is kept in the \(R\) relation. If an entry can be found in \(R\) that has the first unit-label pair associated with the second unit-label pair, then the assignment just made is consistent. Otherwise, the assignment made is not consistent. If the assignment made is not consistent, then another label is chosen for the second unit. If it is consistent, then another unit is chosen in the process repeated. If all the labels for a given unit are exhausted, then the search must backtrack to the first unit-label assignment and try a different label for the unit. This entails unrestricting the \(R\) relation and restricting it to another label for the first unit. The approach just described is a backtracking solution with the constraints of the problem solution kept available in the \(R\) relation. The reduction operators are used to limit the potential choices for future unassigned labels based on the present assignments.

There are two reduction operators available. These are the PHI and PSI operators works by taking entries from \(R\) one at a time and making sure that each entry satisfies certain conditions. If the conditions are not satisfied, then the entry is discarded. There are two parameters supplied with the reduction operators. These parameters are \(k\) and \(p\). The \(k\) parameter specifies the number of free units that must be checked in the test. This is an important proof in the current \(R\) relation. The \(p\) parameter specifies how many sets of free units taken with \(k\) units that must be checked in the test. This proof becomes exponential in the input size of the problem.
necessarily provide more capability for removing entries from $R$ but does show that if $\Phi_{k}^{P}$ does not remove an entry from $R$, then $\Phi_{k+1}^{P}$ does not remove an entry from $R$.

A second reduction operator $\Psi_{i}$ also has the $k$ and $p$ parameters. This reduction operator is used in a slightly different fashion. First, the projection of the $R$ set is formed. The projection of $R$ is the set of all the possible $k$-tuples of unit-label pairs that exist in the $R$ relation.

Then, each projection of $R$ is subject to a test that checks whether all the free units have at least one label such that the combination of the projection entry and the free unit and label is consistent labeling. As $p$ gets large, the power of the operation increases. No statement is made about the $k$ parameter.

The two reduction operators are shown to be equivalent for identical $k$ and $p$ values. The only differences arise when actually implementing the reduction operators.

IV. Inference Engines in RAP

An important features of RAP is the production and analysis of the appropriate inference engines for the flow analyzer, code generator and compactor. For flow analysis, the engine needs to use data paths - i.e., conventional compiler representations for program fragments. A basic optimizable block needs to be recognized. DAGs helped to identify pieces of straight line code and isolate control flow. From this partitioning, we exploit standard compiler optimization strategies specific to hardware generation. For example, block level transformations increase exploitable parallelism. Block merge and/or loop unrolling become natural at compile time.

The Code Generator (CG) inference engine looks at hardware representations as a set of wired functional modules (adders, registers, multiplexers, and multipliers). Wires are legal data paths. Hence, CG mapping of ISDL code segments is guided by these representations in intelligent fashion by using the inference engine to compute the colors of DAG paths and assigning hardware resources to the lowest valences. In distinct departure from current AI integrations, our CG uses inference engine technology for high level synthesis. This is necessary for compiler directed design. Of course the nodes in the DAG represent computations of a partial ordering induced by data dependencies and coloring is by resource allocation. All inference engines must support multiple perspectives, consistency maintenance, search mechanisms, history maintenance, and machine learning. They must reason with default assumptions and heuristics that lead to good design practices. This allows a designer to disagree with RAP conclusions and find better exceptions to the rule-base. Multiple perspectives are necessary for gradual transition from informally based design criteria to formally based paradigms on the manipulated objects. Consistency maintenance is to be performed via a truth-maintenance mechanism to record major decision points.

XV. Conclusions

RAP intelligently aids a VLSI designer to answer the following questions. What ADU(s) and data paths best parallelize code? What is an efficient optimizable code block (from control flow point to control flow point, or mobility of data dependencies)? Given specific scientific application code, what machine graph (hence, VLSI chip design) maps onto a program graph best? How is "best" defined? Given global guidance (e.g., this is a SIMD, MIMD, or ... machine), how can a rapid prototyper with internal inference engines help the VLSI designer match hardware performance to HLL program needs so compiler design is minimized? (Note, many optimizing compilers ignore complex machine instructions. They simply unravel the ISDL and lose all benefits.) Only recently is optimal compilation evident. VSFortran for the IBM 3090 can invoke vector register reuse, complex add/mul, and cache reuse in a partitioned least recently used policy. But even IBM admits that proper management of storage hierarchy drove their design. What actual functional run-times does specific application code take when architectures are fine-tuned? How can a database be built and managed doing code compilation to support trace and debug from source to object code and vice-versa (necessary for an architect)? What compiler criteria should be used to "parallelize" or "serialize" object code? Simple execution time is necessary but not sufficient.
Figure 1 Rapid Architectural Prototyper

Table 1: RI Code Production Comparisons

<table>
<thead>
<tr>
<th>SOLUTION SPACE</th>
<th>MOBILITY</th>
<th>SYSTEMATIC</th>
<th>PROBLEM SOLVING</th>
<th>SPEED</th>
<th>PSEUDORAPID</th>
<th>LEVEL OF EFFORT</th>
<th>COMPILER EFFECTIVENESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Means-Ends</td>
<td>recursive</td>
<td>mixture of forward/backward reasoning</td>
<td>huge</td>
<td>N/A</td>
<td>N/A</td>
<td>Inappropriate (must know solo space a priori)</td>
<td></td>
</tr>
<tr>
<td>Heuristic</td>
<td>fair if tree pruned</td>
<td>&quot;searching to discover&quot;</td>
<td>varies</td>
<td>varies</td>
<td>N/A</td>
<td>Needs brilliant goal function</td>
<td>poor</td>
</tr>
<tr>
<td>Weak</td>
<td>explicit domain-specific knowledge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generator and Test</td>
<td>tend to drift</td>
<td>depth-first</td>
<td>costly</td>
<td>insensitive to span</td>
<td>blind-search</td>
<td>RG for complex architectures</td>
<td>low</td>
</tr>
<tr>
<td>Hill Climb</td>
<td>tend to drift</td>
<td>depth-first</td>
<td>Yes</td>
<td>insensitive to span</td>
<td>inappropiate</td>
<td>RG</td>
<td>RG</td>
</tr>
<tr>
<td>Breadth-First</td>
<td>blind-search</td>
<td>easy to span</td>
<td>insensitive to span</td>
<td>N/A</td>
<td>N/A</td>
<td>Needs brilliant goal function</td>
<td>poor</td>
</tr>
<tr>
<td>Best-First</td>
<td>use DG approach</td>
<td>minimal</td>
<td>minimally sensitive</td>
<td>RG</td>
<td>RG</td>
<td>RG</td>
<td>RG</td>
</tr>
<tr>
<td>Best-First</td>
<td>satisfactory if &quot;rating&quot;</td>
<td>hierarchical knowledge</td>
<td>Yes</td>
<td>RG</td>
<td>RG</td>
<td>RG if arch similar</td>
<td></td>
</tr>
<tr>
<td>Problem Reduction (RNG/OE)</td>
<td>&quot;reasoning backward&quot;</td>
<td>divide and conquer</td>
<td>cycles not permitted</td>
<td>sensitive to goal statements</td>
<td>high</td>
<td>RG if &quot;start&quot; node recognized</td>
<td></td>
</tr>
<tr>
<td>Constraint Satisfaction</td>
<td>selective dependency</td>
<td>directed backtracking</td>
<td>nominal</td>
<td>Yes</td>
<td>RG</td>
<td>RG if used alone</td>
<td></td>
</tr>
</tbody>
</table>

12
REFERENCES


