Viewing Instruction Set Design as an Optimization Problem

Bruce K. Holmer
Computer Science Division[†]
University of California, Berkeley
Berkeley, CA 94720
holmer@vega.berkeley.edu

Alvin M. Despain
Department of Electrical Engineering—Systems
University of Southern California
Los Angeles, CA 90089
despain@cse.usc.edu

Abstract

This paper reviews past attempts to systematize instruction set design and offers an alternative approach. Our technique is based on compaction of microoperations to form instructions. The compaction is done in such a way as to optimize a metric which is a function of cycle count, code size, and instruction set size. To illustrate our technique, optimal instruction sets are derived for data structure creation in Prolog.

1 Introduction

Instruction sets are a necessary and convenient encoding of the operations required to execute application programs. If there were no concern about limiting bandwidth between instruction store and data path, then the best instruction set would be the fully decoded control signals required by the data path. This would allow an uninhibited use of all available hardware in the microarchitecture. In spite of advances in technology for packaging and memory (allowing wider buses and bigger instruction stores), there will always be a point at which the application program size becomes large enough to warrant some kind of encoding of the microinstruction. Compaction of program instructions, however, comes at a cost in performance—limited use of the inherent parallelism of the data path and possible extension of the cycle time due to complex instruction decode. The key to good instruction set design is a balance between performance and instruction encoding.

Methods used for instruction set design have evolved over the years. The initial design for the IBM System 360 instruction set relied heavily on kernels—common instruction sequences of 4 to 40 instructions derived from application and system code [6]. These kernels allowed the benefit of an instruction to be measured accurately. The problem with kernels, however, is that they need to be weighted in importance and can be coded in many different ways. One also tends to forget about other sequences that occur in programs. The Stanford MIPS project [9] took a different approach. They noted that most programming is now done using high level languages. Each instruction, therefore, should be useful to the compiler. In the MIPS project, the person who proposed a new instruction had to modify the compiler to use the new instruction and collect statistics on performance and instruction use. This design approach was later refined by the MIPS company into the 1% rule [13, (page 1-18)]: "any instruction added for performance reasons had to provide a verifiable 1% performance gain over a range of applications or else the instruction was rejected."

The 1% rule tells the designer how to *filter* his trial instructions—allowing only the important instructions to remain. There has, however, been little study of where trial instructions come from. In most design projects, trial instructions come from the designer's experience with previous designs or from the designer's ingenuity. This brings up the question of whether or not there is a systematic way to invent new instructions useful for the application domain of interest.

In this paper we shall review past attempts at systematic instruction set design and present our approach to this problem. In Section 2 we review several approaches taken by other researchers in the past. Then we outline our approach to systematic instruction set design (Section 3) and apply the process to the problem domain of data structure creation in Prolog (Section 4). The final section concludes with an outline for future work.

[†]After 1/1/92: Department of EECS, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208. E-mail: holmer@eecs.nwu.edu

2 Previous Work

There have been many empirical studies on instruction set usage (see for example [17]), but here we will only review some of the more interesting attempts at automating or systematizing instruction set design.

One of the first studies on automatic instruction set design was by Haney [7, 8]. He attacked the problem by creating a model of an instruction set, the generalized instruction set (GIS), which is capable of representing a broad range of the computer instruction sets in use in 1968. Instructions in GIS consist of several fields, some of which have implicit values and are not in the instruction. The user assigns a cost and benefit to each instruction field and a constraint on the total cost. A search program then adds fields one at a time to the instruction format, the one with the maximal benefit first (always remaining within the total cost constraint), until the instruction word is full. By assigning different cost/benefit pairs to the instruction fields, Haney was able to generate instruction sets similar to most in use at the time.

In the 1970's there was an interest in dynamic microprogramming and vertical migration [1, 16, 22]. The basic idea is to move often-used program loops or instruction sequences to writable microcode memory. Each program has its own specialized instructions. Higher performance is obtained by eliminating instruction fetch and decode, by moving heavily used operands into fast registers, and by greater use of data path parallelism. The method we propose is similar, but there are some important differences. The microarchitectures that we consider already eliminate instruction fetch and decode overhead and operand fetch delays through effective use of pipelining and large register files. Our approach derives the entire instruction set, rather than adding new instructions to an existing one. Our instruction derivation takes into account details of the microoperations over all sections of the benchmarks used. Both approaches are similar, however, in that they seek to fully exploit data path parallelism.

A widely held belief during the 1970's was that computer architectures should be customized for high level languages (HLLs). Several studies focused on deriving HLL instruction sets. Bose [4] tackled this problem by specifying a large set of transformations that can be used in various combinations to convert a HLL program into an instruction set. The transformations include aspects of compilation, lexical and semantic analysis, along with transformations on program data representation, storage allocation, and instruction format.

A less ambitious, but very practical study involved revising the Mesa instruction set to better match actual use on a large set of production codes [24]. The goal of the refinement was to improve code density. The instruction set was byte encoded, each instruction consisted of a byte opcode and a set of byte operands (zero, one, or two). A large set of programs were analyzed by first converting their instructions into a canonical form (all instruction arguments were made explicit with a uniform size). Statistics were collected on instruction frequency, instruction pair frequency, and operand value distributions. These statistics suggested new instructions to the designers. New instructions were formed using one of thiee transformations: (1) make an operand value implicit in the opcode, (2) reduce the size of an operand, or (3) combine a pair of instructions into a single instruction. The new instructions were then tested using peephole transformations of the program code, and the process was repeated using statistics from the modified instruction set. The final instruction set gave an overall 12% reduction in code size for the programs studied.

The Mesa instruction set study was done by hand. Bennett [2, 3] automated this process and applied it to generate an instruction set for BCPL. The initial instruction set was derived by hand from a semantic analysis of the necessary operations needed to support BCPL. A new instruction set was generated using the three transformations of the Mesa study. All transformations were tried on all instructions, and the one instance with the greatest predicted code reduction was used. Sometimes the predicted reduction was not correct, resulting in some instructions not being as useful as expected. Overall, however, the process was very effective in reducing the size of the programs studied.

A related area of research is high level synthesis. High level synthesis programs take an algorithm written in a high level language and convert it to a microarchitecture and microcode that satisfies the user's constraints on speed, power, and manufacturing cost. An example of such a system is the JRS IDAS tool set [23] which can handle a wide range of algorithms and target microarchitectures. It is also possible to restrict the microarchitectural framework within which the solution is found. Although this may limit the application domain, it does simplify the design search and potentially allows for better solutions to be found (within the restricted design space). Examples of this approach include ASP [5] and the SCARCE architecture framework [18].

The instruction set design methodology that is presented in this paper is the converse of high level synthesis. Rather than deriving a data path from a specified algorithm or instruction set, our technique starts with a data path model and a set of benchmark programs and then determines the best instruction set for running the benchmarks on the specified data path.

The instruction set design techniques that we have reviewed have several points in common. Each has a model of the instruction set or a set of transformations that move from one instruction set to another. Both, in effect, define the space of all instruction sets reachable by the design process. Each technique also defines metrics with which to judge one instruction set against another. Several techniques have used static code size and few have concentrated on execution speed.[1] This may be due to code size being a more easily measured value than execution speed. Execution speed requires detailed knowledge of the microarchitecture and detailed simulation of benchmark execution, whereas code size does not. In addition, each technique searches the instruction set space by building instructions piece-by-piece (Haney) or by application of transformation rules. Most have used a greedy search strategy—always picking the alternative which gives the greatest immediate improvement in the design metric.

3 Instruction Set Design as an Optimization Problem

As we observed in the previous section, instruction set design, when viewed as an optimization problem, is characterized by: (1) a space of all possible solutions and (2) an objective function or metric measuring "goodness" of a trial solution. Finding the best solution is then a matter of searching through the space of possible solutions and finding the one with the best value of the objective function. In most cases of interest, finding the best solution is not computationally feasible and one must settle for a good (but possibly not best) solution.

Because the space of all possible instruction sets is so large as to be difficult to even conceptualize, we will intentionally limit the scope of this solution space. First we will require that the data path be specified. This will define the microoperations that can be supported by the hardware and will give the time (in cycle counts) required for performing various operations. In addition, we will restrict the data path control circuitry to conform to the opcode control model described below. Specifying both the data path and control will allow the exact enumeration of all possible instructions along with their execution cycle count and interaction with other instructions. In addition, the cycle time can be factored out of the performance equation, because the instruction set is defined by the contents of opcode ROMs and decode PLAs in the control circuitry. The details of this control model will be presented below.

A basic concept behind our search strategy is to cast instruction set design as a variation on microcode compaction. Primitive operations supported by the data path are combined together into instructions given constraints on data dependencies, number of instruction bits available to specify the operations, and the number of distinct instruction opcodes available. There are many ways of combining microoperations into instructions, so we use a set of benchmark programs and performance metrics to judge whether one instruction set is better than another.

In the following subsections we will present our instruction set metrics and pipeline control model. We will then give a general overview of our search strategy.

3.1 Instruction Set Metrics

Casting instruction set design as an optimization problem depends on finding a good metric for measuring the quality of one instruction set against another. Certainly the ultimate metric would involve implementing the instruction set (and data path and control) in hardware and measuring the performance, power consumption, manufacturing cost, and any other factor important for its intended application. This process is quite unreasonable given that the search through the space of instruction sets may require looking at millions of alternatives. Practicality dictates that our metric be an easily computed approximation to "building and measuring." The exact metric will depend on the requirements of the application. In our work, we will be biased toward cycle count, since this is the typical emphasis in computer design. Our metric is a function of:

cycle count (*C*) The total number of clock cycles required to execute the benchmark set. This represents the performance of the processor.

static code size (*S*) The total number of words required to store the instructions used to represent the benchmark set. This represents the instruction set architecture efficiency. It has an indirect effect on the performance by affecting the instruction cache performance.

instruction set size (*I*) The total number of distinct instructions required to represent the benchmark set. It is also the same as the number of distinct instruction opcode values. This represents the complexity of the instruction set architecture. It has an indirect effect on the control hardware cost and the design verification complexity.

One of the simplest metrics would just be cycle count. The best instruction set would give the minimal cycle count for the benchmark programs. There are many solutions using this metric, however, since one can add any new instruction to the best instruction set and the cycle count would remain the same. This limitation can be overcome by specifying that the best instruction set is the *smallest* instruction set with minimal cycle count.

[1] Here we are referring to only the attempts at automation. Many hand-derived instruction sets have used execution speed as a major criterion.

Instruction sets deemed "best" by the minimal cycle count metric still have a major flaw. Some of the instructions in the instruction set may be used only once or twice in order to reduce the cycle count by a few cycles in obscure cases. The 1% rule avoids this difficulty and eliminates those instructions which are used only for seldomly occurring optimizations.

If we assume a constant cycle time, then the 1% rule can be expressed more concisely as

$$100\,\Delta C/C + \Delta I < 0. \tag{1}$$

The addition of an instruction to the instruction set ($\Delta I = 1$) must be counterbalanced by more than 1% drop in cycle count ($\Delta C/C < -1/100$).

The above equation is a difference equation, and is not satisfactory as an objective function in the search for an optimal global solution. To motivate the form of the appropriate function, we can consider the differences to be infinitesimals and take the definite integral. The best instruction set would then be the one that minimizes

$$100 \ln C + I. \tag{2}$$

This function has the desirable property of being invariant (to within a constant factor) to scaling of C.

Modifications to the 1% rule are possible. If one wishes to constrain static code size, then a term could be added for code size. For example, if our rule states that a new instruction will be accepted if it causes a 5% static code size reduction with no change in cycle count, then the analogous equation to Equation 2 is

$$100 \ln C + 20 \ln S + I. \tag{3}$$

The metrics presented here are just some of the possibilities. Other metrics have been given by Bose [4]. In our example in the next section we will use the minimal cycle count and 1% metrics.

3.2 Pipeline Control Model

There is a whole range of possible microarchitectures that could be used to implement an instruction set. The pipeline control models that we will consider are variations of data stationary control [14, (pages 118–119)]. In data stationary control, the instruction word (or opcode) passes through the pipeline in parallel with the data that it operates on and is decoded at each stage into control signals.[2] Some versions of data stationary control allow the instruction (or instruction opcode) to be modified as it passes through the pipeline. The simplest data stationary control model, *single-cycle pipelined*, allows only single-cycle instructions and is comparable to the RISC I architecture [19]. Our example in the next section will also make use of the single-cycle pipelined model.

[2]This is logically what happens—the implementation is optimized for circuit area and speed.

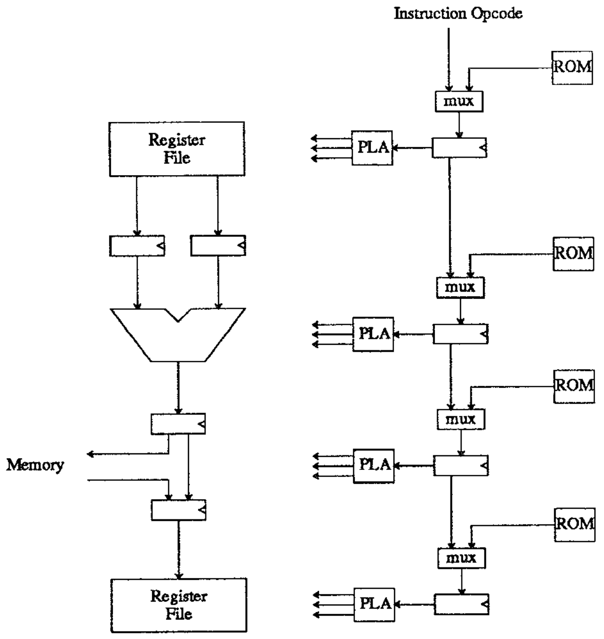

Figure 1: Opcode Pipeline Control Model

If one desires more flexibility in instruction set design, however, a more sophisticated pipeline control model is necessary. Greater flexibility can be obtained by dynamically injecting "internal" opcodes into the standard data stationary control pipeline. The use of internal opcodes is illustrated in Figure 1. On the left side of the diagram is a data path with pipeline latches between four stages: register read, arithmetic operation, memory operation, and register write. The register file at the top and bottom of the pipe are the same multi-ported register file—the loop has been unrolled to make it more readable. The diagram is also simplified in terms of the number of pipeline stages. A real implementation might have pipeline latches for each clock phase with two phases per cycle. Also not shown are details like the bypassing logic needed to forward results of one instruction to the instructions behind it in the pipeline. On the right side of the diagram, the control pipeline is partially shown. Besides the opcode pipeline, there are pipelines for the register specifiers and other instruction fields. The process of injecting internal opcodes into the normal flow of external opcodes is accomplished using the multiplexors and ROMs at each stage of the opcode pipeline. Note that the internal opcodes are not available to the programmer as individual instructions.

The internal opcodes can be injected in one of two basic ways: insertion or replacement. During opcode insertion no external opcodes are lost. As internal opcodes are inserted into the stream of external opcodes, the opcodes behind the insertion point are stalled. Opcode replacement, however, involves the replacement of an external opcode by an internal opcode. There is no stalling of the following opcodes. In general, multicycle instructions are implemented by inserting opcodes at the first pipeline state, whereas conditional instructions (conditional branches, conditional loads, etc.) are implemented by replacing opcodes at the second or third stage of the pipeline. The internal opcode control model is actually used in several special and general purpose microprocessors [12, 20, 21].

Restricted versions of the internal opcode model are possible. For example, if only insertion of opcodes at the first stage of the pipeline is supported, then we have the Clipper's macro instructions [10]. If we can only insert instructions that are available to the programmer into the first stage of the pipeline, then we have the Atlas' extracodes [15].

3.3 Searching the Space of Instructions

Although we are making several simplying assumptions in the process of instruction set design, there is still a large gap to span from the benchmark programs and data path to the final instruction set.

One possible search technique is to enumerate all possible instructions that the data path and control can support.[3] Any instruction set is a subset of this enumerated set of possible instructions, so the search for a good instruction set can proceed by moving instructions into or out of the current instruction set, revising the benchmark code using peephole rules, and computing the change in the instruction metric. We will refer to this process of searching the instruction set space the *enumeration* method.

In the example given below the instruction set is "discovered" by the process of assembling microoperations into single-cycle instructions. There was no attempt at pre-computing all single-cycle instructions possible with a given data path. We will call this technique of assembling instructions from microoperations the *compaction* method, because of its strong relationship with microcode compaction techniques.

The example given below can be solved using branch-and-bound search because the benchmarks are small and the problem domain limited. For more typical applications a more flexible process is needed, at the expense, however, of not being able to determine optimal results. Figure 2 gives an outline of the compaction method of instruction set formation. The benchmark programs are first compiled into a graph of primitive (or micro) operations. These microoperations are determined by the data path specification. The graph

[3]In practice, some type of limiting factor on each instruction must be used, such as cycle count or flow graph size.

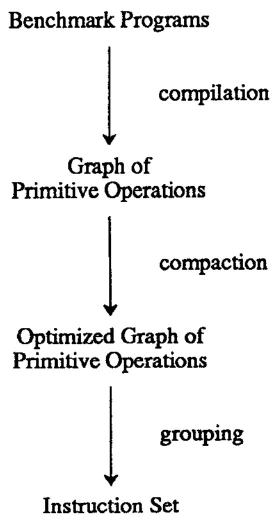
Benchmark Programs
compilation
Graph of Primitive Operations
compaction
Optimized Graph of Primitive Operations
grouping
Instruction Set

Figure 2: Compaction Technique of Searching the Instruction Set Space

of operations is then rearranged and compacted in order to improve the performance metric (cycle count, graph size, etc.). Techniques for rearrangement and compaction are well know in the field of microcode compaction, some of which are branch-and-bound, list, trace, and percolation scheduling. Microoperations in the optimized graph are then grouped into instructions. Both the compaction and the grouping must be done with consideration given to instruction formation. A potential worst case is that each part of the optimized graph gives rise to a distinct instruction, yielding an unmanageable instruction set size. The compaction process must be constrained by the needs to limit instruction set size. This can be achieved with the appropriate metric, the 1% rule, for example, which balances instruction set size against performance.

4 Example: Data Structure Creation in Prolog

To illustrate our method for instruction set design, we will use an example that actually came up in the design of the VLSI-BAM, a general purpose microprocessor with extensions for the support of high performance execution of Prolog [12]. During the instruction set design of the VLSI-BAM, our research group had proposed several instructions for the support of data structure creation in Prolog's execution model (also called write-mode unification). Initial versions of the results given in this section helped us decide on the final instruction set support.

4.1 Benchmark Programs

The hypothetical example in Figure 3 illustrates how Prolog creates data structures. The Prolog execution model maintains several stacks in memory. We are concerned with only two: the heap (also called the global stack) and the environment (the procedure activation record). Data structures are created on the heap; however, pointers to the data may also be placed in the environment or the register file. Prolog is a dynamically typed language, so data is tagged with its type. In this paper, we will assume that the tags are placed in the most-significant four bits of the 32-bit word. Data structures consist of two or more tagged data words. The tagged data can be any of the following: a constant (`functor/4`), a tagged variable pointer (shown with a `V` tag), a word from the registers (`reg(0)`), a word from the environment (`env(0)`), or a tagged list or structure pointer (not shown). As the structure is created, the top-of-heap pointer, `H`, is incremented.

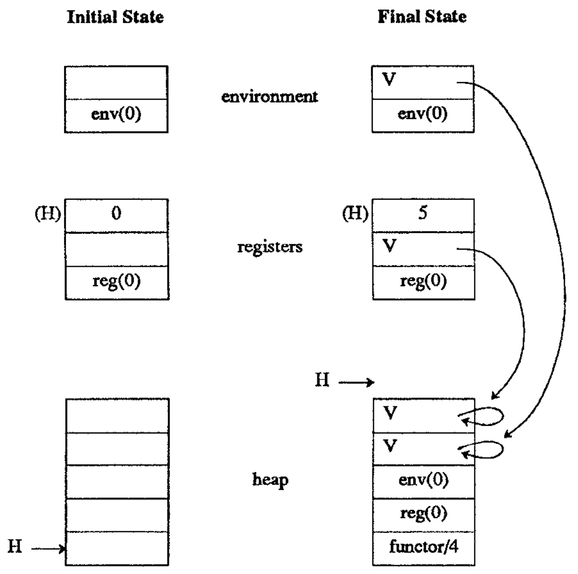

Figure 3: Example of Data Structure Creation in Prolog

Our goal is to find an instruction set that is good for creating these kinds of data structures. Note that this data structure creation requires no decisions to be made in the computation, so the programs will consist of straight-line code and the resulting instruction sets will have no branch or jump instructions. Recall that the inputs to our instruction set design procedure are a data path and a set of benchmark programs. The data path will be given below, and our benchmarks will be a set of data structures specified as initial and final state pairs. For the examples in this paper we used 43 different two word structures and 43 three word structures. This set of data structures represents a nearly exhaustive set of all two and three word structures. Ideally we would weight each benchmark proportionally to the frequency with which it occurs in actual Prolog programs, but for illustration we employ unweighted benchmarks.

4.2 Model of Data Path

The model for the data path that we will use is based on the VLSI-BAM [12]. Instead of using the full internal opcode control model employed there, however, we will restrict ourselves to the ***single-cycle pipelined*** control model. The optimal instruction set will change as a function of the parameters of the data path. These parameters include the number of ports to the register file or memory. Figure 4 shows the model with the parameters most similar to the VLSI-BAM. The two word port to and from memory supports the load or store of two consecutive memory locations. Note that several details of a real data path have been abstracted away.[4] For example, we will assume that the interconnect boxes allow all possible interconnections of buses. In a real design, only some of the connections would be present.

4.3 Details of the Instruction Set Derivation

Figure 2 illustrates the general outline of instruction set derivation. In this particular example we did both the compilation and compaction by branch-and-bound search, and the grouping was into single-cycle instructions.

For each benchmark data structure, branch-and-bound search is used to find the sequence of microoperations requiring the fewest cycles that will take the initial state of registers and memory to the final state. Because we are assuming a single-cycle pipeline control model, instructions are formed by making each cycle's set of microoperations be an instruction. A pattern matching is done on the microoperation list to generalize each instruction field value. For example, if the microoperations for a cycle reference register 1 and register 2, these references are generalized to register i and register j. If the microoperations reference register 1 multiple times, all of these references become register i.

The search is constrained by data dependencies and data path resource limits. There is a single cycle delay between loading data from memory into a register and using that data in a computation. This restriction

[4]Removal of details is not necessary, but is done here to prevent unnecessary complication of the example.

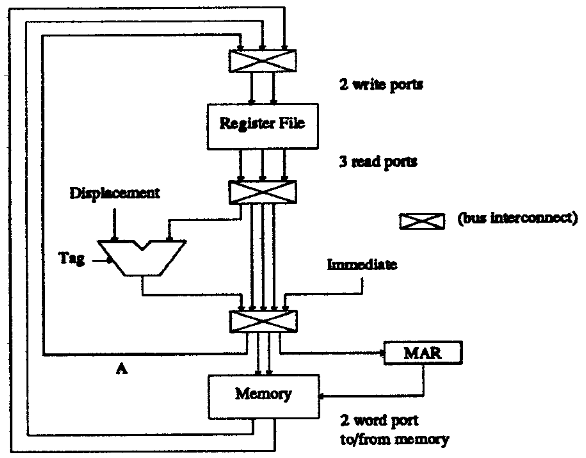

Figure 4: Model of Data Path for the VLSI-BAM [12]

is typical of pipelined RISC processors. For our examples, we will also limit the number of register writes, not originating from memory, to be one per cycle (this corresponds to the path labelled with "A" in Figure 4).[5] Another important constraint is the number of instruction bits available for encoding the operations to be performed. Instructions will be fetched over a dedicated instruction bus, and one 32-bit instruction word can be fetched each cycle. In this example, no implicit values of the instruction operands will be allowed. The number of bits for each instruction field is given in Table 1. These values were picked to match those used by the VLSI-BAM [12]. Each instruction has one opcode field, but the use of the other fields is constrained only by the total number of bits needed by all of the operations in the instruction.

The final instruction set is the "union" of the instruction sets derived from each benchmark. The merging of instruction sets takes into account the case in which one instruction is a more general case of another instruction. In this case the less general instruction is discarded and the more general instruction is retained.

Most benchmarks have multiple solutions with minimal cycle count. In this case each minimal solution is tried separately in the union process. The union with the fewest instructions is the final instruction set. To illustrate, Figure 5 shows the solutions for three benchmarks. Benchmarks 1 and 2 have a single optimal solution but benchmark 3 has two optimal solutions. Solution 2 is picked since it will give the smallest set of instructions.

The result of the search and union is an instruction set which is optimal over all benchmarks for the minimal cycle count metric. To derive the solution for the 1% metric, an additional step is required. This step involves removing one or more benchmarks from the union process in an attempt to reduce the size of the final instruction set. The resulting instruction set is then used to compile all of the benchmarks to determine the total cycle count. More details can be found in [11].

[5]This constraint can be relaxed, but at the cost of increasing the number of forwarding paths necessary.

Table 1: Number of Bits Required by Each Instruction Field

instruction field	*# bits*
opcode	6
register	5
tag	5
displacement	5
immediate	16

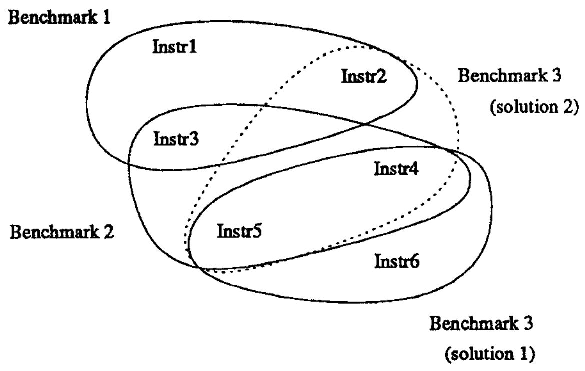

Figure 5: Example of Instruction Set Formation

Table 2: Optimal Instruction Set for Simple RISC Data Path

Ri → mem(Rj) Rj+disp → Rj	146	*push*
tag ‖ (Ri+disp) → Rj	90	*tag address*
imm → mem(Ri) Ri+disp → Ri	69	*push immediate*
mem(Ri+disp) → Rj	30	*load*
Ri → mem(Rj+disp)	26	*store*

These results are for the minimal cycle count and 1% metrics. The second column gives the number of times the instruction is used in the benchmarks. The parameters in the data path model are: 2 register read ports, 1 register write port, 1-word data port width, and 1 ALU. To execute all 86 benchmarks a total of 363 cycles is required.

4.4 Results

Using the benchmarks and data path model described above, we have derived optimal instruction sets for both the minimal cycle count and 1% metrics. For the simplest instance of the data path model (fewest number of resources) the resulting instruction set for both the minimal cycle and 1% metrics has 5 instructions and is shown in Table 2. The instruction set has no real surprises. Data is moved to and from memory using *load*, *store*, and *push*. The *push* instruction allows a 5-bit increment value and some benchmarks use a non-unit increment. Tagged pointers are created using the *tag address* instruction (the register transfer notation *tag* ‖ *exp* represents the insertion of *tag* into the most significant bits of *exp*). Constants are obtained using the *push immediate* instruction. We should note that the solution is not a trivial result—it is unique and was chosen from 20 instructions considered by the search out of a total of 124 possible instructions. Note that these instruction sets are not sufficient for general purpose calculations. This is due to our constrained data path model and benchmark domain.

If we allow more bandwidth to the register file and memory, then the optimal instruction set becomes more complex. The optimal solution for the minimal cycle count metric is shown in Table 3. The data path parameters for this example closely match those of the VLSI-BAM processor. Part of the instruction set is identical to the instruction set for the simple data path, and the additional instructions are double word versions of *load*, *store*, and *push*. The *push* instruction is dropped and compensated by heavier use of *store*. The *load immediate* instruction is also included and is used for loading a constant into the register file in preparation for a double word push of the constant and another value. This result guided the instruction selection for the VLSI-BAM.

Table 3: Optimal Instruction Set using Extra Ports to Register File and Memory

Ri → mem(Rj+disp)	101	*store*
tag ‖ (Ri+disp) → Rj	90	*tag address*
imm → mem(Ri) Ri+disp → Ri	64	*push immediate*
Ri/Rj → mem(Rk) Rk+disp → Rk	36	*push double*
mem(Ri+disp) → Rj	26	*load*
imm → Rj	5	*load immediate*
mem(Ri+disp) → Rj/Rk	2	*load double*
Ri/Rj → mem(Rk+disp)	2	*store double*

These results are for the minimal cycle count metric. The second column gives the number of times the instruction is used in the benchmarks. The parameters for the data path model are: 3 register read ports, 2 register write ports, 2-word data port width, and 1 ALU. To execute all 86 benchmarks a total of 330 cycles is required.

The VLSI-BAM instruction set includes all of the instructions in Table 3 except *push immediate* which is replaced with *add immediate* and *store immediate*. The optimal instruction set for the 1% metric simply eliminates the *load double* and *store double* at a cost of four additional cycles. The search considered 40 instructions out of a total of 799 possible instructions.

If we add a second arithmetic unit to the data path, then the instruction set becomes even more complex. Table 4 lists the optimal instruction set using the 1% metric for a data path with two arithmetic units along with extra bandwidth to the register file and memory. Several of the instructions do two logically independent operations (for example, *load with increment*). Other instructions do a sequence of operations which require multiple cycles in the previous data paths. For example, two instructions create a tagged pointer and store it with other data (*push double register/tagged address* and *push double tagged register/register*). Two instructions store a tagged pointer in both the register file and memory (*store and remember tagged address* and *store and remember tagged register*). Note that the two store and remember instructions cannot be subsumed by a single more general instruction because this more general instruction[6] would not fit in the 32-bit instruction word.

[6] tag ‖ (Ri+displ) → mem(Rj+disp2)
tag ‖ (Ri+displ) → Rk

Table 4: Optimal Instruction Set using Extra Ports and Two ALUs

imm → mem(Ri) Ri+disp → Ri	66	*push immediate*
tag ‖ (Ri+disp1) → mem(Rj+disp2)	60	*store tagged address*
Ri → mem(Rj+disp1) Rj+disp2 → Rk	45	*store with add immediate*
mem(Ri+disp1) → Rj Rk+disp2 → Rk	30	*load with increment*
tag ‖ (Ri+disp) → Rj	16	*tag address*
Ri/(tag ‖ (Rj+disp1)) → mem(Rj) Rj+disp2 → Rj	16	*push double register/tagged address*
tag ‖ (Ri+disp) → mem(Ri+disp) tag ‖ (Ri+disp) → Rj	14	*store and remember tagged address*
(tag ‖ Ri)/Rj → mem(Ri) Ri+disp → Ri	7	*push double tagged register/register*
tag ‖ Ri → mem(Rj+disp) tag ‖ Ri → Rk	4	*store and remember tagged register*
imm/Ri → mem(Rj)	3	*store double immediate/register*

These results are for the 1% metric. The second column gives the number of times the instruction is used in the benchmarks. The parameters for the data path model are: 3 register read ports, 2 register write ports, 2-word data port width, and 2 ALUs. To execute all 86 benchmarks a total of 261 cycles is required.

5 Conclusions and Future Work

In this paper we have presented an overview of a new systematic technique for instruction set design. This technique generates an instruction set when given a data path, a control model, and a benchmark set. The optimization metrics are based on simple combinations of cycle count, static code size, and instruction set size. When we applied this technique to the design of an instruction set for data structure creation in Prolog, we were able to use branch-and-bound search to derive optimal results for each metric and data path.

The example given in this paper brings up the question of whether the derived instruction set can be effectively used by a compiler. In a sense, using branch-and-bound search provides us with a "perfect" compiler. In the past, most problems with instruction set design have been due to using poor compilers. An instruction set designed using a poor compiler will often inhibit the success of a new optimizing compiler. If a "perfect" compiler is used, then the instruction set will become a better match for the production compiler as the compiler is improved.

Future work with our technique will include its application to complete Prolog programs. The derived instruction sets will include arithmetic, branch, and jump instructions. Using the full internal opcode control model will allow conditional instructions which can possibly replace instruction idioms containing conditional branches (for example, *unify immediate* in [12]). Common instruction idioms can be captured using multi-cycle instructions when using the modified 1% rule (Equation 3). Using the data path of the VLSI-BAM, the automatically derived instruction set for complete Prolog will be directly compared to the VLSI-BAM instruction set.

6 Acknowledgments

We are grateful for the helpful comments from Mike Carlton, Peter Van Roy, and the anonymous referees. We are especially grateful for discussions with David Culler.

This work is partially funded by the Defense Advance Research Projects Agency (DARPA) and monitored by the FBI under contract No. J-FBI 91-194. Additional support was generously provided by NCR and ESL.

References

[1] A. M. Abd-Alla and D. C. Karlgaard. Heuristic synthesis of microprogrammed computer architecture. *IEEE Transactions on Computers*, C-23(8):802–807, Aug. 1974.

[2] J. P. Bennett. Automated design of an instruction set for BCPL. Technical Report 93, University of Cambridge, Computer Laboratory, 1986.

[3] J. P. Bennett. *A Methodology for Automated Design of Computer Instruction Sets.* PhD thesis,

University of Cambridge, Computer Laboratory, 1988. Also available as Technical Report 129.

[4] P. Bose and E. S. Davidson. Design of instruction set architectures for support of high-level languages. In *11th Annual International Symposium on Computer Architecture*, pages 198–206, 1984.

[5] W. R. Bush, G. Cheng, P. McGeer, and A. M. Despain. An advanced silicon compiler in Prolog. In *Proceedings of ICCAD*, 1987.

[6] D. Gifford and A. Spector. Case study: IBM's system 360—370 architecture. *Communications of the ACM*, 30(4):292–307, Apr. 1987.

[7] F. M. Haney. *Using a Computer to Design Computer Instruction Sets.* PhD thesis, Carnegie-Mellon University, 1968.

[8] F. M. Haney. ISDS—a program that designs computer instruction sets. In *Fall Joint Computer Conference*, pages 575–580, 1969.

[9] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, and T. Gross. Design of a high performance VLSI processor. In *Third Caltech Conference on VLSI*, page 33, 1983.

[10] W. Hollingsworth, H. Sachs, and A. J. Smith. The Fairchild CLIPPER: Instruction set architecture and processor implementation. Technical Report UCB/CSD 87/329, University of California, Berkeley, Computer Science Division, 1987.

[11] B. K. Holmer and A. M. Despain. Optimal instruction sets for Prolog structure creation. Technical Report ACAL 91-4, University of Southern California, Advanced Computer Architecture Research Laboratory, 1991.

[12] B. K. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, W. R. Bush, A. M. Despain, J. M. Pendleton, and T. Dobry. Fast Prolog with an extended general purpose architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 282–291, 1990.

[13] G. Kane. *MIPS RISC architecture.* Prentice-Hall, 1989.

[14] P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill Book Company, 1981.

[15] S. H. Lavington. The Manchester Mark I and Atlas: A historical perspective. *Communications of the ACM*, 21(1):4–12, Jan. 1978.

[16] P. S. Liu and F. J. Mowle. Techniques of program execution with a writable control memory. *IEEE Transactions on Computers*, C-27(9):816–827, Sept. 1978.

[17] A. Lunde. Empirical evaluation of some features of instruction set processor architectures. *Communications of the ACM*, 20(3):143–153, Mar. 1977.

[18] J. M. Mulder, R. J. Portier, A. Srivastava, and R. in 't Velt. An architecture framework for application-specific and scalable architectures. In *16th Annual International Symposium on Computer Architecture*, pages 362–369, 1989.

[19] D. A. Patterson and C. H. Sequin. A VLSI RISC. *Computer*, 15(9):8–18, Sept. 1982.

[20] J. Pendleton, S. Kong, E. Brown, F. Dunlap, C. Marino, D. Ungar, D. Patterson, and D. Hodges. A 32-bit microprocessor for Smalltalk. *IEEE Journal of Solid State Circuits*, SC-21(5):741–749, Oct. 1986.

[21] J. M. Pendleton. Private communication about the design of a new SPARC implementation, 1990.

[22] T. G. Rauscher and A. K. Agrawala. Dynamic problem-oriented redefinition of computer architecture via microprogramming. *IEEE Transactions on Computers*, C-27(11):1006–1014, Nov. 1978.

[23] D. W. Runner and E. H. Warshawsky. Synthesizing Ada's ideal machine mate. *VLSI Systems Design*, pages 30–39, Oct. 1988.

[24] R. E. Sweet and J. G. Sandman, Jr. Empirical analysis of the Mesa instruction set. In *Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 158–166, 1982.