A Tool for Processor Instruction Set Design

Bruce K. Holmer
Department of EECS
Northwestern University
Evanston, IL 60208-3118 USA
holmer@eecs.nwu.edu

Abstract

This paper presents a systematic technique for generating new instruction sets which are optimized for a given microarchitecture and set of benchmark programs. This process consists of the following steps: generation of execution traces, formation of code segments, optimal recompilation of the code segments to produce candidate instructions, and covering of the instructions from the code segments to yield the final instruction set. To illustrate the use of the new technique, an instruction set is generated for the execution of compiled Prolog programs.

1 Introduction

Even though some may say that the current instruction set architectures are sufficient for almost all application needs (both general and special purpose), there has been no lack of recent introduction of new instruction sets or revisions to current instruction sets.

This paper presents a tool for assisting the design of instruction sets of instruction set processors. The tool takes as input a data path and a set of benchmarks (representing the application domain) and produces as output an instruction set which optimizes a metric. The metric includes terms for cycle count, code size, and instruction set size. This process partially automates the process that is currently done manually: (1) performance simulation, (2) determination of common code sequences, (3) finding the proper encoding of these sequences (instructions), and (4) removing instructions that do not contribute enough to the metric being optimized. The process presented is useful for either instruction set based processors or for vertical microcode machines. In both cases, limited instruction bits prevent the full use of the possible parallelism in the data path. The proper choice of instruction set minimizes any loss in the potential performance.

Designing a programmable computer involves the iterative refinement of the instruction set and data path implementation. A change in the instruction set prompts a modification to the data path and changes in the data path may require changes to the instruction set. The basic concept behind the method of instruction set synthesis described here is to cast the process 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 bits available to the instruction, and the number of distinct instruction opcodes available.

Although a large part of the design effort for a new processor is the data path, it is assumed in this paper that this is done by hand or by a high-level synthesis system. It is assumed that the data path is held fixed, and the best instruction set for that data path and benchmark set is derived. The tool can be incorporated into a manual or automated cycle of data path modification and instruction set derivation.

There have been several previous attempts at automatically or systematically generating instruction sets [4, 5, 8]. None has become a commonly used technique. Most likely this is because the metric being optimized is not closely related to actual performance, and performance is very important for most applications.

Perhaps the technique most closely related to the one presented in this paper is vertical migration [1, 3, 16]. The basic idea of vertical migration is to move often used functions, program loops, or instruction sequences to writable microcode memory. 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. Many of these benefits, however, can be obtained by using a data path which eliminates instruction fetch and decode overhead and operand fetch delays through effective use of pipelining and large register files.

Vertical migration identifies possible additions to existing instruction sets which can be added using writable microcode or implemented with a coprocessor. The technique presented in this paper is different because it can generate a new instruction set as a whole. Furthermore, it forms instructions by rearranging microoperations into one or two cycle "packages". Vertical migration, however, builds "big" instructions by combining a sequence of instructions already present in the instruction set.

Recent CAD research has also considered instruction set

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specic permission. (c) 1994 ACM 0-89791-687-5/94/0009 3.50

design [2, 13, 15, 17]. The work in this paper offers another alternative method for finding good instruction sets, and it emphasizes the trade off between instruction set size and performance.

A detailed description of the new method is given in Section 2. In Section 3 the technique is applied to the design of an instruction set specialized for Prolog. Conclusions and future research are outlined in the final section.

2 Automatic generation of instruction sets

The new technique for instruction set design is composed of three major steps. First, the benchmarks are transformed into a set of code segments weighted by dynamic execution frequency. Second, the code segments are recompiled using the microoperations of the specified data path. A cycle's worth of microoperations becomes an instruction. (If multicycle instructions are allowed, then the microoperations of two, three, or more cycles may make up the instruction.) Lastly, the instructions from one recompilation for each code segment are combined together to form the final instruction set. Because each code segment has many possible recompilations, search is performed to find the selection of recompilations which minimizes the optimization metric.

Besides a model for the data path and control, a model for the compiler is also assumed. The compiler uses a simple code generator with a sophisticated peephole optimizer [6]. Recompilation of code segments is essentially just finding peephole optimization rules where one is allowed to create one's own instructions!

Decomposing the benchmarks into small segments of code allows search to be used to find all of the near optimal compilations for the code segment. This process of compiling the code segment (at the microoperation level) creates the instructions that will go to make up the final instruction set. Trying alternative instruction sets is now reduced to trying different ways of covering the recompilations of the code segments. Because many of the important recompilations are computed before covering, alternative instruction sets and their performance can be quickly examined.

The complete process is illustrated in Figure 1. The following subsections will cover each of the steps in this process: compilation of the benchmarks, execution trace generation, formation of code segments, optimization of code segments, and instruction set formation. Before discussing these steps, however, the metric used to judge between alternative instruction sets is presented.

2.1 Metrics for instruction set generation

The success of automatically deriving instruction sets depends on finding a good metric for measuring the quality of one instruction set against another.

In this paper the bias is toward execution time, but the metric could also incorporate code size (important for em-

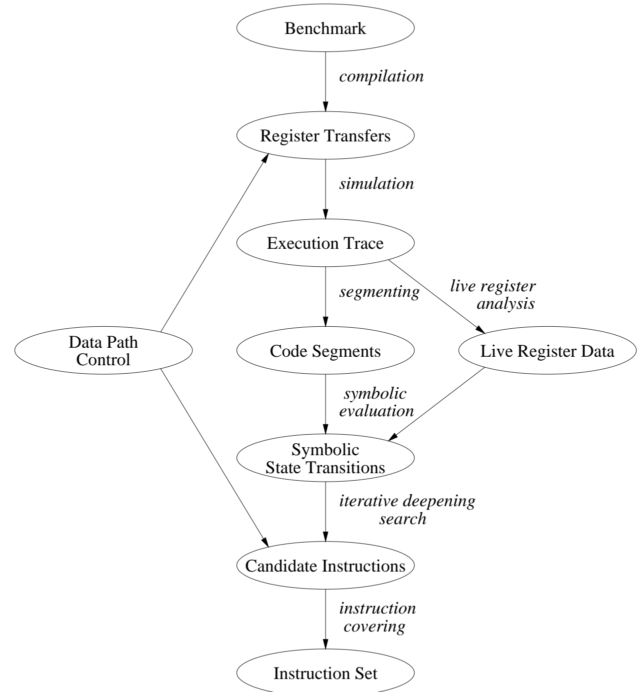

Figure 1: Overview of Instruction Set Derivation

bedded applications). Because the instruction set generation is done for a fixed data path, cycle time can be factored out of the performance equation. This allows cycle count to be used as an estimate of the performance. The pipeline control model allows modifications to the instruction set to be implemented as changes in the content of the control PLAs. No significant change in the delay through the PLAs will be caused by changes to the instruction set. For more detail about the control model used see [9].

Besides minimizing cycle count, it is important to make sure that each instruction is contributing to the performance. Each instruction represents additional commitment of design and verification time and use of precious opcode space. The measure of an instruction's contribution to the performance is the percent performance loss if the instruction is removed from the instruction set.

The metric proposed here requires that each instruction contributes at least N% to the total performance. By varying the value of N one can achieve the desired instruction set size. The MIPS R2000 instruction set was designed with a 1% metric in mind [12, page 1.18]. Use of the 1% metric for a Prolog processor gives rise to a relatively small instruction set (see Section 3).

The "N% metric" should satisfy the relationships

$$\begin{aligned} f(I,C) &< f(I+1,C), \\ f(I,C) &< f(I,C+1), \\ f(I,C) &= f(I-1,(1+N/100)\,C), \end{aligned} \quad (1)$$

```
;; branch has one delay slot
;; branch annuls when not taken
8e9c:  bne,a  4,r2,8f28
8ea0:  ld     8(r1),r3
....
8f28:  mov    r2,r4
```

```
if (r2 == 4)
    pc:  8ea4
else
    pc:  8f2c
    r3:  mem(r1+8)
    r4:  r2
```

Figure 2: Transformation of Code Segment

where I is the number of instructions in the instruction set and C is the cycle count of the benchmarks. The metric is an increasing function of both instruction set size and cycle count and balances a decrease in I of one with an increase in C of $N\%$.

These properties imply the functional form

$$f(I, C) = I + \alpha \ln C. \quad (2)$$

Substituting this functional form into the equality in Equation 2 and solving for α yields

$$\alpha = 1/\ln(1 + N/100) \approx 100/N. \quad (3)$$

Code size can be included into the metric

$$f(I, C, S) = I + \alpha \ln C + \beta \ln S, \quad (4)$$

where S is the code size and β determines the percent change in the code size required before adding a new instruction (assuming no change in the cycle count). By changing the values of α and β one can obtain instruction sets which are optimized for performance, code size, or some combination of the two. Also, the final instruction set size can be adjusted by increasing or decreasing α and β.

2.2 Formation of code segments

The benchmark programs are compiled and simulated to produce an execution trace which contains information about the operation types, instruction addresses, register accesses, and memory accesses. Each trace is stored and later used for live register analysis and code segment selection.

Even though Figure 1 shows the data path register transfers as the target of compilation and as the building block of the code segments, actually any convenient instruction set can be used. Preferably the instruction set is one for which a simulator or trace generation tool exists. Once the symbolic state expressions are formed for each code segment, then very little of the instruction set originally used remains.

The goal of code segment generation is to sample typical code sequences and to weight them by their dynamic execution frequency. The benchmark programs are broken into segments of code by randomly selecting points in the execution trace which are used as starting points for the code segments. Random selection is used to eliminate any systematic bias that may be introduced by boundaries selected by non-random methods. The end of the code segment is selected so that each code segment will require three or four execution cycles on the target data path.

Code segments are not restricted to be within a basic block. They often contain branches and jumps. In these cases, only the instructions along the execution pathway are included in the code segment. The segment does contain stubs for the conditional branches exiting it. This is somewhat analogous to how trace scheduling goes beyond basic blocks [7].

There are alternatives to segmenting the program into code segments. For example, simulated annealing can be used on the flow graph of the benchmarks to do both operation scheduling and instruction formation [11]. More experiments are necessary to decide which is the best alternative—each has its advantages. For example, segmenting's main disadvantage is its limitation on the distance of operation movement (an operation is stuck in the segment that it is in), but in a sense this limitation on code movement is just what is required to allow the peephole optimizer to effectively use a manageable peephole window size. An advantage of code segments is that the recompilation of each code segment is independent from the others and hence is trivially and highly parallelizable.

2.3 Optimization of code segments

Each code segment is converted into declarative form by symbolic execution. This declarative form is a list of symbolic values for the program state at the end of the code segment expressed in terms of the state at the beginning of the segment. When conditional operations or branches are present, then a separate symbolic state is given for each of the execution pathways through the code segment. Live register information derived from the execution trace can be used to identify dead registers in the code segment. Eliminating dead registers can simplify the symbolic representation of the code segment.

Figure 2 illustrates the transformation of a code segment into symbolic form. The original code segment can be expressed using register transfers or instructions from a known instruction set. The example contains a conditional branch, so the code segment follows one of the possible execution pathways. The conditional branch also introduces a conditional in the symbolic expression. In the example, when `r2` is not equal to 4, the final value of `r3` is the value in memory at address `(r1+8)` where `r1` is the value of register 1 at

the beginning of the code segment.

For each symbolic state transition, heuristic 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. This search is done using iterative deepening—first a single cycle solution is attempted, then if that does not work, a two cycle solution, and so on until a solution is found. The search is constrained by data path resource limitations, the control model, and the number of bits available for the instruction. If single-cycle instructions are desired, then each cycle's set of microoperations becomes an instruction.

The search for optimal recompilations of the code segment is done by symbolically executing the register transfers supported by the data path. Whenever a register is referenced or a constant is used, then space is allocated in the instruction word. The search backtracks whenever the operations require more bits than the instruction word contains or more resources than the data path supports.

In this paper it is assumed that an instruction is fixed-length and made up of a selection of field types which include the opcode, immediate data, register specifiers, and displacements. These fields are fit into the instruction word or take implicit values determined by the opcode.

After the code segment is successfully recompiled, a pattern matching is performed on the register transfer specification of the instructions to generalize each instruction field value. For example, if an instruction references register 1 and register 2, these references are generalized to register i and register j.

Immediate and opcode field sizes can vary and their sizes are determined during the final instruction set formation. The actual value of the immediate needed by the instruction is used to determine the minimum number of bits needed to represent this value. The data path description specifies what kinds of encoding are supported for immediates (for example, zero extension or sign extension).

2.4 Final instruction set formation

The goal now is to combine the code segment recompilations to form the final instruction set. The final instruction set is the union (or cover) of instructions using one recompilation for each code segment. Each code segment recompilation consists of a cycle count and a list of instructions formed by the microoperation sequence. Depending on which recompilations are included in the cover, one trades off instruction set size and total cycle count. The combination of code segment recompilations which minimizes the N% metric is the desired cover.

To illustrate, Figure 3 shows three code segments, each of which has two solutions (recompilations). Each solution is represented by a row and each instruction by a column. A dot indicates that the instruction is required by the solution. The three rows outlined can be combined to form the instruction set containing instructions 1, 3, 5, and 6 and requiring 13 execution cycles. If the other solution is used for code segment 1, then the number of instructions in the instruction set increases by one, but the cycle count decreases to 12. The 1% metric favors the larger instruction set because the performance gain is more than 1% $(4 + 100 \ln 13 > 5 + 100 \ln 12)$.

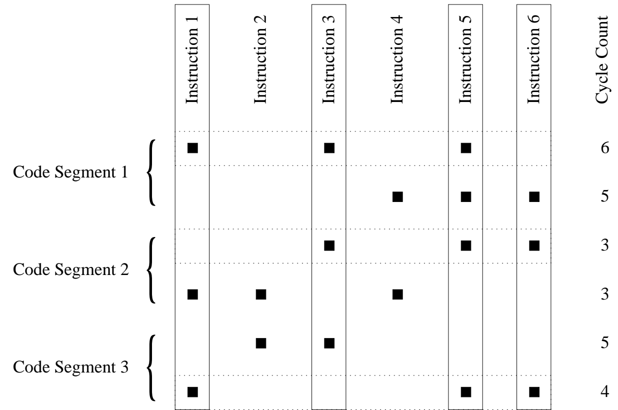

Figure 3: Example of Instruction Set Formation

There is another important consideration that must be dealt with during the instruction covering, and that is subsumption—when one instruction is a more general version of another. An instruction is subsumed by another when its immediate field is smaller than the other or when register specifiers are less general. For example, `addi Ri,Imm,Ri` is subsumed by `addi Ri,Imm,Rj`.

Instruction subsumption is important because the final instruction set should not contain both an instruction and an instruction which it subsumes. To prevent this from happening, when an instruction is added to the cover, a check is made for subsumption. Any instructions in the cover that are subsumed by the instruction being added are removed.

The application domain may require that the instruction set be complete. That is, the instruction set must have the ability to compute any function. This property can be ensured by picking a set of operations known to be complete and creating special "completeness" code segments that each contain a single operation from the complete set. These completeness segments are included with the code segments derived from the benchmarks. Because the final instruction set must by definition cover every code segment, the final instruction set supports the completeness segments and therefore is itself complete.

Note that the covering process can be modified so that the new instruction set is a strict superset of some given instruction set. This would allow one to search for the best additions to an established instruction set.

Adding and deleting instructions from an established instruction set is a common occurrence in the evolution of general purpose instruction sets (for example, the PowerPC deletes some instructions, and recently the Sparc and MIPS have added instructions). Allowing a limited number of both additions and deletions to an established instruction set can easily be incorporated into the optimization metric. The instruction set size, I, in this case would be redefined to be the number of differences between the reference instruction set and the generated instruction set (that is, the number of deleted instructions plus the number of new instructions).

3 An instruction set for Prolog

In this section the techniques described in Section 2 are used to generate an instruction set specifically optimized for the execution of Prolog benchmarks. The goal is to compare an automatically generated instruction set with the hand-crafted instruction set for the VLSI-BAM processor [10]. This instruction set is chosen because it has been implemented as a VLSI chip, and details of the implementation are available allowing the automatically generated instruction set to be based on the same data path.

The pipeline consists of five stages, each with a pair of transparent latches activated on opposite phases of the cycle. The register file contains 32 word-sized registers and has two read and two write ports. One of the two read ports is double-word width allowing the read of two consecutive registers. There are separate busses to the instruction and data caches, and the write-back data cache design requires a one cycle pipeline stall when a store is followed immediately by either a load or store. A load followed by immediate use of the data also requires a one cycle pipeline stall. Branches and jumps are delayed and have a single delay slot.

The data path supports four immediate types: a four-bit tag, a zero-extended immediate, a sign-extended immediate, and a sign-extended immediate with a four-bit tag. The size of the immediates is determined by the instruction set generation process.

The data path is controlled using a variation on data stationary control [14]. This model allows the opcode passing through the pipeline to be dynamically changed, which can be used to easily implement instruction annulling (cancelling the next instruction in the pipe) and conditional execution (changing the instruction's operation based on the condition code or a comparison earlier in the pipeline).

For more detail about the data path and control models used by the VLSI-BAM and this instruction set study see [9].

Four medium-sized Prolog benchmarks were used as input to the design technique. These benchmarks were used in the instruction set studies for the VLSI-BAM [10].

Using the VLSI-BAM data path and benchmarks as input, 34 instructions were automatically generated. This instruction set is the result of minimizing the 1% metric while covering the compilations for 1029 code segments. The 1029 segments include the 29 completeness code segments and the 1000 segments sampled from the four benchmarks. The code segment compilations produced 2372 different instructions from which the 34 in the final instruction set are chosen.

For the results in this paper, the covering of the code segment solutions was done using a greedy algorithm. The initial step finds an instruction set which covers all of the code segments. For the instruction set described here, this initial instruction set contains 99 instructions. The next phase attempts to remove instructions one at a time. This change in the instruction set is acceptable if the value of the 1% metric improves. Deletion of an instruction can cause some of the code segments to no longer be covered. These code segments are recompiled using the trial instruction set, and the new recompilation results are included as additional input to the covering process.

Space limitations prevent the inclusion of a complete description of the generated instruction set here. The most important differences of the instruction set with a more traditional hand-generated instruction set are listed here.

- The generated instruction set has no immediate forms for any arithmetic instructions (except add immediate).
- Because compiled Prolog code tends to consist of many short basic blocks which often end with a short jump instruction, the generated instruction set chose to combine short forward unconditional jumps with common operations such as compares, loads, and stores.
- The compiled code also performs a sizable number of register-to-register moves. This is done to set up the arguments for the next subroutine call (the subroutine arguments are passed in the register file). The resulting instruction set combines register-to-register moves with jump, compare, and memory instructions.
- Even the subroutine call instruction does not escape the 1% metric. By itself, removing call from the instruction set causes about a 2% degradation in the performance (due to now needing two instructions: jump followed by load immediate or jump followed by a read PC instruction). The instruction generation found that this loss can be compensated by now being able to combine the jump with an adjacent register-to-register move to form a single instruction. This compound jump and move instruction is also very useful for other coding situations.
- The VLSI-BAM uses an unsigned maximum operation to find the top of a stack which interleaves two separate stacks. When an allocate is done for either stack, the true top of stack is the maximum of the two separate top of stack values. The derived instruction set chose a different approach. Using two instructions, a compare

followed by a conditional move, the same performance can be achieved.

Another instruction set was generated using a different sample of code segments taken from the four benchmarks. The vast majority of instructions in the resulting instruction set are identical to those in the first instruction set generated. The differences are mainly due to instructions right on the border of 1% performance benefit being included in one instruction set but not the other.

Using the 1% metric results in an instruction set with 34 instructions (compared to nearly 60 for the VLSI-BAM). Performance estimates show that the derived instruction set is about 3 to 4% slower than the VLSI-BAM, but this performance could be regained by simply using a smaller percent cutoff (for example, a 1/2% metric). The instruction set would become larger but the performance would improve and match that of the hand design.

4 Conclusions

In this paper a novel technique for automating the instruction set design process has been presented. It is based on performance measures for benchmarks using defined information about the data path (resource limitations, instruction word size, and control model). The technique can be used for both the generation of new instruction sets and for finding the most important modifications to an existing instruction set.

When the design technique is applied to a pipelined data path supporting compiled Prolog execution, non-trivial suggestions for both instruction inclusion and deletion are obtained. Compound instructions which combine an arithmetic or memory operation with a short jump or register-to-register move are common. For Prolog, immediate forms for arithmetic instructions are not important enough to include in the instruction set. And the call instruction (jump and link) can possibly be replaced (suffering no loss in performance) with a compound jump/register-to-register move.

The technique for instruction set design presented here can potentially be a valuable aid for designers of specialized processors. The technique searches for common combinations of microoperations without any preconceived opinions of what "good" instructions should look like. The process can suggest instruction possibilities that may not have been thought of before, and it will rigorously remove instructions of marginal benefit.

Acknowledgements

Mike Smith, David Gilbert, and the anonymous reviewers provided helpful comments. This work is partially funded by the National Science Foundation Research Initiation Award MIP-9210692.

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] A. Alomary, T. Nakata, et al. PEAS-I: A hardware/software co-design system for ASIPs. In *Proceedings of EURO-DAC'93*, pages 2–7, Sept. 1993.

[3] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3), Mar. 1993.

[4] 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.

[5] P. Bose. *Instruction Set Design for Support of High-Level Languages*. PhD thesis, University of Illinois at Urbana-Champaign, 1983.

[6] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, Oct. 1984.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

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

[9] B. K. Holmer. *Automatic Design of Computer Instruction Sets*. PhD thesis, University of California, Berkeley, 1993.

[10] 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.

[11] I.-J. Huang and A. M. Despain. Synthesis of computer instruction sets. In *Proceedings of the 31st Design Automation Conference (DAC)*, 1994.

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

[13] H. Kitabatake and K. Shirai. Functional design of a special purpose processor based on high level specification description. *IEICE Trans. Fundamentals*, pages 1182–1190, Oct. 1992.

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

[15] C. Liem, T. May, and P. Paulin. Instruction-set matching and selection for DSP and ASIP code generation. In *Proceedings of EDAC-ETC-EUROASIC*, pages 31–37, 1994.

[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] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the 7th International Symposium on High-Level Synthesis*, May 1994.