Adaptive Compilation for Complex Uniprocessor Hardware

Katja Daumueller and Darko Stefanovic

December 1994

Project Summary

Instruction scheduling and register allocation are potentials for either bottlenecks or speeding up the compiler. It has been shown that in order to produce most efficient code, register allocation and instruction scheduling are best executed simultaneously. In order for them to be fast, however, the compiler has to exploit useful architectural features, too. Therefore, for optimal performance, a close intertwining of instruction scheduling, register allocation and the use of architectural features is necessary.

For many architectures, creating optimal code is intractable, and only simple architectures allow optimal performance. Therefore, several approaches have been developed which use heuristics. These approaches may shorten the compile time, but create suboptimal code and may not be tailored for the used architecture. As architectures change, retargetable compilers have to devise a new intertwining for every new machine, which could take a long time.

Therefore, the objective is to implement an adaptive learning strategy for register allocation and instruction scheduling. With this strategy, less time has to be spent with the development of new, good heuristics, because there exists a simple mechanism for testing. Also, heuristics can be tested for their adaptation to the application environment.

The methods used are an architectural specification language and an adaptive learning strategy. The project also entails developing a simulation tool, which simulates the timing behaviour of a real machine from the architectural description. A learning strategy is developed, whose input are several heuristics and different programs. It learns the best heuristics for the given programs and chooses the best parameters for the heuristics. The learning relies on the compiler back-end, which compiles the programs and gives timings with the architectural simulation tool. For the compiler back-end, tools are investigated which create back-ends, given the architectural specification, the instruction set and heuristics for register allocation and instruction scheduling. For experiments, however, the compiler back-end is expected to be hand-written.

The project is significant because it puts architecture as a parameter in the optimization of register allocation and instruction scheduling. Heuristics for new machines can be developed faster, because they are evaluated immediately. Also, new machines can be compared regarding their adaptability to compilation strategies. Heuristics can be developed specifically for a set of application programs. By incorporating profiling into optimization, special cases which occur seldom in practice are given less weight, and the compiler is made more efficient for everyday use.

Project Description

Abstract

The problems of compiling for the current generation of high-performance uniprocessors are outlined: complex machines with multiple pipelined units and multiple instruction issue require sophisticated code generation, especially register allocation and instruction scheduling, to exploit their performance potential. However, the fast pace of development of new technology does not allow time for handcrafting complicated code generators. We describe a model for developing hardware and its compilers together, wherein part of the task of code generator generation—selection and tuning of heuristics—is accomplished through an adaptive learning mechanism.

1 Introduction

The current trend in uniprocessor architectures is towards complex implementations. Although the basic RISC model is preserved, the organization of the processor is no longer simple as in early RISC designs. Multiple execution units are present, with deep pipelines. For various reasons this trend can be expected to persist in the foreseeable future. Exploiting this potential parallelism requires cooperation from the compiler. The compiler must ensure that processor resources (in particular, execution units and the register file) are used in a balanced manner, so that delays and stalls are avoided and issue rate is brought close to maximum.

We envisage a system for coordinated development of architecture designs and their compilers; it is based on a formal description of architectural parameters and on cooperation between the compiler and an architecture model/simulator tool.

Within this system, we intend to concentrate on the design of an *adaptive* compiler. The code transformations that a compiler must perform in order to exploit the complex hardware efficiently and not merely correctly can be seen as an exploration of a vast (in some cases, infinite) search space of valid code fragments. Existence of good heuristic algorithms for problems such as register allocation and instruction scheduling for older and current architectures suggests that this search space is structured in ways that allow efficient approximate solutions. We want to find out if it is possible to determine this structure exploratively. In particular, adaptive search techniques (also known as *learning*) may be used to this end.

2 Related Work

In the literature, some approaches deal with register allocation [Cha82, BCKT89, CH90, San90, CK91], some with instruction scheduling [GM86, GR90, War90, BR91] and some try to perform register allocation and instruction scheduling simultaneously or try to integrate it best [Kar84, Mor91, BEH91, PF91, BHE91]. Two approaches were found where the machine was described with parameters or with a model language [BEH91, BR91].

2.1 Register allocation

Concerning register allocation, in all of the approaches, heuristics are applied, because otherwise the problem would be intractable.

One basic approach [Cha82] (Chaitin's algorithm), for example, builds an interference graph, where each edge represents an interference between two live ranges. Chaitin's allocator attempts to color the graph in k colors, where k is the number of registers available. If it cannot find a k-coloring, it spills one or more live ranges in order to transform the graph into one that has a k-coloring. Hereby, in order to test if the graph can be k-colored, nodes whose degree (the number of neighbors) is less than k are removed from the graph. If there does not exist any node with degree less than k, spill code has to be inserted, and the process has to be repeated with a new graph. The algorithm runs in time linear in the size of the expression tree.

Briggs *et al.* [BCKT89] improve this method by removing the nodes in a special sequence and applying another criterion. After all nodes have been removed, k-coloring is applied to the nodes in the order they had been taken out of the graph. In case a node cannot be colored, it remains uncolored, and at the end, spill code is inserted for the uncolored nodes. This approach avoids spilling in many cases, because it removes nodes out of the graph in a certain order, and it only assigns spill code at the end of the coloring process.

The algorithm of Chow and Hennessy [CH90] also uses coloring as a central paradigm. The algorithm uses a priority function, which assigns values for every live range in a basic block. If a live range interferes with one of highest priority which was just assigned, it needs to be split. The algorithm runs before code generation. If spilling is necessary, registers are always spilled to their home locations. The claims around this algorithm are that it uses less memory and processing overhead than Chaitin's approach, and suffers rarely from worse allocation due to its higher granularity.

Callahan and Koblenz [CK91] give another approach for register allocation. The argumentation is that longer pipelines and longer latencies between the time an operation is issued and the time it is available will bring significance to the placement of spill code. In the algorithm, the program's loop and conditional structure is represented as a tile tree. Using a graph coloring approach similar to Chaitin's, spill candidates are singled out in each tile. Final spill decisions are made according to control flow information (*e.g.,* where along the tile tree should the variable be spilled) and using statistics. The algorithm claims to have superior performance with inline expansion and procedure calls.

Santhaman [San90] describes a method for compiling programs using interprocedural register allocation. A strategy for handling programs built from multiple modules is presented, as well as algorithms for global variable promotion and register spill code motion. The approach uses profiling information.

Most of these approaches claim to run in size linear of the reference tree. The quality of the code, however, suffers from not considering the specialities of the hardware and not performing instruction scheduling at the same time. Pure instruction scheduling algorithms, however, may not result in optimal register allocation, and spill code could be introduced.

2.2 Instruction scheduling

The technique of Gibbons and Muchnick [GM86] reduces the number of pipeline interlocks significantly, and runs in time $O(n^2)$ for a basic block of n instructions. This is implemented by sweeping down a dag whose nodes are instructions within one basic block. In order to schedule, it is important to know if instructions will cause interlocks with subsequent instructions. As a lookahead is too expensive, heuristics are used. Information for these heuristics can be kept in the node, *e.g.,* length of the longest path from the instruction to a leaf node.

Improvements for global instruction scheduling for pipeline processors are described by Golumbic and Rainish [GR90]. A number of structural transformations of loops and branches are introduced, for example exchanging **if** and **then** parts of conditional statements because one branch has a delay which cannot be filled by the corresponding **if** or **then** block.

Waren [War90] describes a scheduling algorithm in basic blocks on the IBM RS/6000 processor, which is designed to use arithmetic units as efficiently as possible. First, the paper describes the constraints of the machine, for example that the fixed and floating point units operate in parallel and are each capable of executing one instruction per cycle. Scheduling is done (alternately for fixed and floating point operands) from a dependency graph, which also keeps information about the maximum total delay along the path of each instruction to the end of the basic block. In every iteration, an operation is scheduled depending on the maximum total delay, ‘liveness weight’ (*e.g.,* choose loads later because they increase register usage) or ‘uncovering’ (pick the instruction which causes the most subsequent instructions to become ready). The scheduling algorithm is done before and after the register allocation phase.

Bernstein and Rodeh [BR91] aim at global (intra-loop) scheduling, which uses the control and data dependence information summarized in a Program Dependence Graph, to move instructions beyond basic block boundaries. In the scheduling algorithm, instructions with the biggest delay in a basic block, and after that with the biggest critical path (a measure of how long it takes to complete all instructions which are in the same basic block as the instruction in question) are scheduled first within a basic block. Globally, basic blocks are scheduled according to their topological order in the control flow graph. Speculative processing is used globally to speed up the algorithm. The algorithm aims at superscalars as well as VLIW machines, and a parametric description of the respective superscalar RISC processor provides details. In the description, for example the kind of functional units and their number can be specified for a the superscalar RISC processor.

Heuristics found in this investigation of previous literature can always be used and tested in our approach, and it can be tested how well they fit to different architectures. Instruction scheduling has one major heuristic, which is scheduling the instruction with the longest delay in the basic block first.

2.3 Combining register allocation and instruction scheduling

The instruction scheduling algorithms respect the hardware constraints, in general but in order to produce good code, register allocation constraints must be considered, too. Attempts to have interactions between the register allocation and instruction scheduling phase are made in several papers.

Karr [Kar84] introduces a coagulating code-generator, which has a tight coupling of register allocation, instruction selection and code optimizations. It makes use of profiling the code. A flow graph is obtained from the intermediate representation, and its arcs are labeled according to the frequency with which the nodes on their ends occur. Then, instruction selection and register allocation is done in the flow graph. Arcs with the highest frequencies are then selected and tested for conformity on exit and entry nodes, and because of this, busy parts of the program are treated first. If nodes on the endpoints of the arcs do not agree, minor cost repairs are undertaken, such as inserting copy instructions or revising storage allocation. Morris [Mor91] augments this method by improving the register allocation, code ordering, general storage allocation and by substantially simplifying cohabitation (occurrences the compiler has allocated to the same storage, but which have intersecting live ranges) resolution. This approach is close to our approach in that it considers profiling information. It does not, however, deal with architectural constraints, and only allows the profiling information to be used in one heuristic.

Proebsting and Fischer [PF91] introduce an algorithm which optimizes register allocation as well as instruction scheduling for pipeline architectures with load delays of one; for higher delays, the algorithm serves as a good heuristic. For trees, the algorithm runs in time linear in the number of nodes in the expression tree, and dags are split up so that they can be processed as trees. Also, optimal positions for register spilling can be predicted. The algorithm is similar to the Sethi-Ullman (SU) algorithm, and first finds the SU minReg values. Second, it determines the operation and load orders (similarly to the SU algorithm), and third it emits the instructions in canonical order. The approach is restricted in that it cannot handle register variables, and it is assumed that all operations decrease the number of registers by one. For our purposes, this approach would be too restrictive because delays are only allowed for loads and not for e.g. branches.

Bradlee *et al.* [BEH91] compare what benefit coupled scheduling of register allocation and instruction scheduling has. It investigates three algorithms, of which the first completely separates register allocation and instruction scheduling. The second emulates communication by encouraging the scheduler to keep register pressure at a level conducive to good register allocation. Finally, the third algorithm provides a vehicle through which the register allocator and the instruction scheduler can communicate their register requirements. Results show that in benchmark tests, both the second and third algorithm show significant improvements over the first algorithm, however the third algorithm is only slightly faster than the second. Modifications on the third algorithm are suggested which could yield improvements, but the conclusion is that expensive approaches like the third one which closely couple register allocation and instruction scheduling, are unnecessary.

Finally, Bradlee *et al.* [BHE91] discuss a retargetable code generator (Marion) which uses a description language for RISCs. The description language describes kinds of registers, stages of the pipeline and specific constraints on instructions in the pipeline. In the mapping from the intermediate representation to the machine code, some C code has to be written by the user. For instruction scheduling and register allocation, the three algorithms of [BEH91] are implemented. A common heuristic for scheduling is again the maximum distance heuristic, where the node with the maximum distance along any path from the instruction node to a leaf node in the code DAG is scheduled. New techniques have been developed in Marion to deal with scarce resources on the same cycle (structural hazard), pipeline delays because of unknown branch addresses (control hazard) and explicitly advanced pipelines and temporal registers because of sub-operations.

Marion has the benefit of having an architecture description language. Its scheduling algorithms are deterministic and do not take into account profiling information. Therefore, they cannot adapt to a new architecture optimally, or to specific usage patterns. It only can incorporate new heuristics in Marion's scheduling system, which would be provided by our system. Nevertheless, it is the closest approach to our system because it integrates instruction scheduling and register allocation and has a description for pipeline architectures.

Most of the introduced systems lack a detailed architecture description which would make their system retargetable. Some do not integrate register allocation and instruction scheduling, and for some, the register allocation and instruction scheduling cannot be augmented with heuristics.

3 Scientific Questions

The question we are ultimately interested in is:

> *How can one efficiently build/maintain good optimizing compilers in the face of rapid hardware development?*

To begin answering this question, we pose several narrower questions: What means are necessary to describe the structure and functioning of modern uniprocessor hardware? Can a generic code generator produce good code for a range of architectures and their implementations? If not, can code generator generators be used effectively?

In particular, we are interested in adaptive strategies where the effort of learning to produce good object code lies with the automated system rather than, or not exclusively, with the human compiler writer. The first specific question to be answered is: can an adaptive compiler learn good instruction scheduling heuristics for basic block scheduling? If so, how well does it perform in comparison with existing schedulers and how far is it from the optimum? What is the training effort? How do the answers depend on particular architecture and implementation?

Thereafter, the problem will be broadened to global (intraprocedural) scheduling and to simultaneous register allocation. Finally, the problem of complete code generation will be examined.

4 Approach

Here we outline first our outlook on the task of compiler construction for modern hardware, and then focus on the subtask of adaptive compilation which we intend to tackle first.

The general structure of the proposed codesign methodology is sketched in Figure 1. The drawing is meant to indicate the elements of the system; their relationship is discussed in the text.

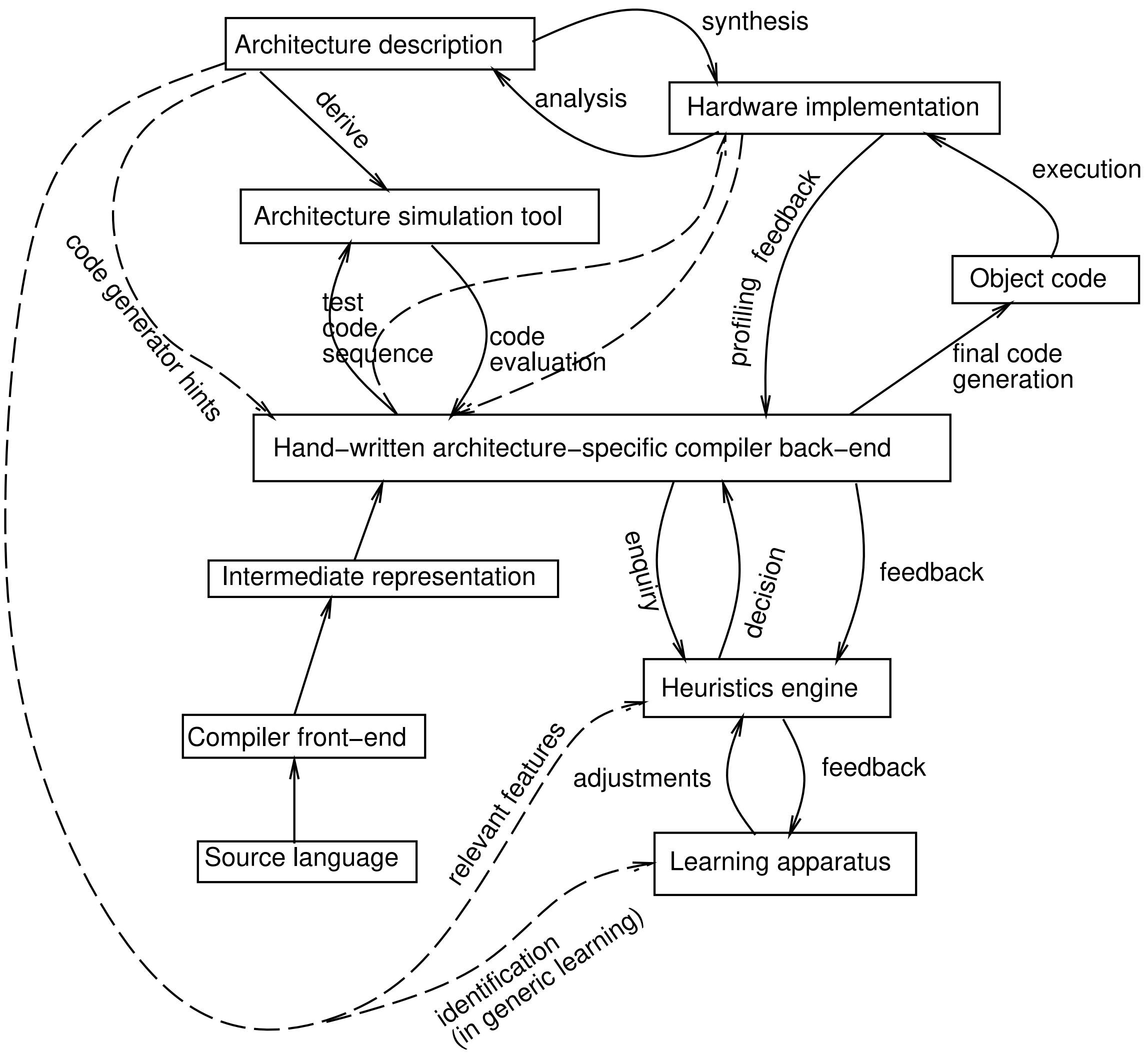

Figure 1: The envisaged system development structure

4.1 Hardware description

The starting point is a description of a hardware architecture and organization (implementation). The description specifies the functionality and timing of the hardware components. The description is formal; a special language is used for the purpose. The description has two parts: a higher-level architecture description and a lower-level implementation description.

The *architecture* description gives the instruction set, register set, addressing and other elements common to all implementations of an architecture.[1] The level of detail in the architecture description must allow accurate simulation of the effects on the architecturally specified processor and memory state of executing single-stream object codes, in the absence of exceptions or context switches.

The *implementation* description is specific to a given implementation of an architecture, that is, a given chip set. It gives the functional units of the processor, their pipeline stages and latencies. It also describes the memory subsystem (caches, write buffers, address translation buffers etc.) and its latencies. The level of detail in the implementation description must allow accurate timing relative to clock speed—in addition to providing correct end results.

The description language will be designed to permit varying the level of detail provided, to confront the possible difficulties in obtaining complete specifications of existing machines, as discussed below. It will be possible to elaborate the descriptions in a modular way. There have been several proposed languages with a similar purpose, and ours will most closely resemble the machine description language in the Marion system [BHE91], but it will not include a "compiler writer's virtual machine", as we do not want to constrain the range of compiling strategies.

4.2 Hardware implementation

An existing hardware implementation may be analysed to obtain a hardware description, or the description can serve as the specification for hardware synthesis. The architecture specifications provided by hardware vendors are usually quite complete, so deriving a formal description is straightforward. However, the literature for particular chip implementations is scant, partly because the implementations change frequently, partly because the information is withheld as proprietary. Arriving at complete and veracious descriptions may therefore require considerable experimental effort. Going in the other direction, from a detailed formal hardware description to its realization is a

[1] In particular, it captures the resource dependences between instructions, needed for instruction scheduling, and perhaps sufficient for naïve instruction scheduling.

course we can recommend to hardware developers, and which they to a considerable degree already follow. Although this direction is essential to the idea of coordinated codesign of new hardware and its compilers, it must remain outside the scope of this proposal. Once the research proposed here is underway, we intend to approach one or more hardware manufacturers, in cooperation with whom we would have both the resources and a natural setting to explore the codesign problems as well.

4.3 Architecture simulation tool

In recent years, several newly introduced computers have suffered in the marketplace because they were not object-code compatible with any previous hardware and did not have compilers for the new hardware ready in time. Although this problem afflicts radically new architectures more than traditional uniprocessor designs, it is worth solving by developing simulation tools ahead of the hardware implementation. Once a formal description of an architecture is given, since it was posited that it describes the functionality of the architecture, it should be possible to automatically derive a simulation tool therefrom. This simulator is capable of simulating the execution of code sequences, or programs in general, and produces the ordinary results of execution. In addition, given an implementation description, the simulator can produce the performance statistics for the various hardware components; in particular, it can return the timing for executing a given code sequence under given initial conditions. How such a tool can be derived automatically, and how it can be efficient are two important problems to be solved.

The description has both declarative and procedural elements, and the simulation tool interprets both procedurally to the extent desired. For example, under certain circumstances timings may be computed (more or less accurately) without a complete simulation. In general, just as the description itself can be modularly expressed with greater or lesser detail, so can the level of detail, complexity and accuracy of the simulation be adjusted as needed.

4.4 Compiler front end

Some well-known source language is translated using well-established techniques into some intermediate representation appropriate for optimizations and for final code generation. The choice of source language is not critical to this project, but for concreteness and for utility to others we will use C. Rather than develop a new front end, we will use either the `lcc` system or `gcc` and adapt it to our needs. Thus developing the front end will be relatively straightforward.

4.5 Compiler back end

The back end of the compiler lies at the heart of the proposed system. It is responsible for optimizations over the intermediate representation and for code generation. Although it is entirely possible to derive a correct back end automatically from a formal specification of the intermediate representation and the architecture description, this is outside the scope of the proposed work. Instead, the back end is hand-written. It interacts with the rest of the system in some ways that are not found in traditional compilers.

The code generator must often solve the problem of choosing among (numerous) different code sequences, each of which is correct and performs the same net computation. The choice therefore must be made according to the predicted efficiency of the sequences. Two examples are instruction selection, and, perhaps more appropriate for current machines, instruction scheduling. The proposed compiler can invoke the architecture simulation tool and obtain a timing prediction for a code sequence (or larger fragment) under consideration (See Figure 1, solid arrows). Whenever the hardware is designed to allow it, the compiler can directly execute the code sequence [Bak91] and obtain a timing measurement (Figure 1, broken arrows). Finally, the compiler can use profiling data from the execution of complete programs. The preceding criteria are based on some form of black-box performance prediction. In addition, the compiler can accept a qualitative characterization of the architecture if it is provided in the description. This takes the form of code generator hints (Figure 1, broken arrow).

The problem of generating optimal code is not computable in general, and even many important restrictions known as compiler optimizations are at best computationally intractable. For example, it is clear that optimal instruction scheduling of n instructions may require testing $n!$ code sequences in the worst case. Although there has been work on optimal compilation through exhaustive search (for instruction selection [Mas87]), it appears necessary in many cases to restrict the search. In order to exploit the assumed or observed structure of the large problem space intelligently, heuristics are used. Heuristics are often expressed as *ad hoc* rules with intuitive justification, such as the rule for alternating low-conflict and high-conflict instructions in RISC scheduling [GM86]. Such rules tend to be hardwired into the optimization algorithms with unclear separation between the mechanisms and the policies, between rules for correctness and rules for optimality. We propose that a separate module—the *heuristics engine*—should be present to control the heuristics; the compiler poses enquiries and accepts decisions from the heuristics engine, and passes feedback information from the several sources mentioned above. In this manner, the rôle of the heuristics is isolated and flexibility improved: one can supply a null heuristic engine and still obtain correct code, or, on the other hand, one can develop a sophisticated heuristics engine which may be largely independent of the particular optimization task.

Heuristics can be viewed as ways of pruning a search space by making local decisions where a global optimum is sought. The efficacy of this approach depends on the structure of the search space. We are not aware of previous studies of this structure for important optimization problems except as estimates of worst-case costs or observation of achieved cost with heuristics. An investigation of the search space for basic block instruction scheduling is currently underway.

4.6 Heuristics engine

This component encapsulates the heuristics, so that the compiler is not burdened with code implementing arbitrary and spurious decisions. There are many kinds of heuristics that may be used. In a transformational context (for example a tree rewriting system), the heuristics can be formulated either as absolute rules (which always succeed when their preconditions are met), or as numeric criteria (which compete against each other). In all contexts, there will be heuristics expressed as numeric parameters controlling other rules or fixed mechanisms (often in the form of weights controlling an evaluation function). A goal of the project is to determine the exact nature of the rules and parameters the heuristics engine must support for the class of heuristic search problems found in compiler optimizations.

Once the heuristics have been separated from the compiler, it becomes possible to make them *adaptive*, or self-improving.

4.7 Learning engine

Heuristics embody the understanding of the structure of hard problems that goes beyond the expression of provable optimality criteria. For this reason they cannot be derived deductively from the problem description. Usually the human compiler-writers draw them from their experience and justify by plausible inference. There are two drawbacks to this, caused by increasingly complex and fast-changing hardware. First, the immense complexity of the problem space, such as in the problems of code generation for current and forthcoming RISC designs, may cause human implementors to lose track of some interactions between hardware components. In other words, the understanding and intuition may not carry over to exceedingly complex hardware. Secondly, with the 2-year cycle of chip implementations, there is simply not enough time to hand-tune heuristics to new processors. Therefore, it is well to investigate the possibility of automating the process of heuristics learning.

To that end, we propose to attach a learning engine to the heuristics engine. This module will accomplish the task of adaptation of the heuristics. It will be largely generic in nature, operating without an understanding of the meaning of the heuristics. There is a considerable body of work in machine learning; some of the newer techniques can, with appropriate modification, be used in the compiler optimization problems [LM88, IAS+]. Clearly, modifications are needed to match the structure of the problem, its search space and the form of heuristics.

4.8 Scope of work proposed

The full system as drawn in Figure 1 requires cooperation from experts in several fields: compiler design, computer architecture, circuit design, machine learning, and language design. It is a large undertaking appropriate for industrial research and development. However, our rôle will be to do experimental work on several subproblems, with much less investment, and verify the feasibility of the approach. In particular, fixing the architecture to an existing one, and constructing a simple but reasonably accurate model for it, we will be able to test the adaptive compilation part on one or more optimization tasks. Now that we have outlined our vision of modern compiler design, we will concentrate on the work we propose to undertake immediately, namely the adaptation (learning) of heuristics for code generation.

The first experiment will examine scheduling instructions within a basic block, as a separate phase occurring after register allocation. Since this problem is computationally intractable, heuristics are used. Initially, the implementor provides a set of heuristics, and the task of the adaptive compiler is to learn the relative relevance of using each heuristic in the set to the goal of minimizing schedule cost.

More specifically, the experiment will be conducted as follows. A compiler constructs basic blocks to be scheduled as the last phase in code generation, or a specially constructed tool extracts basic blocks from existing executable programs for a certain machine. Another specially constructed tool simulates the execution of a straight-line instruction sequence on that machine, calculating the number of cycles the execution takes. A third tool computes the architecturally defined scheduling constraints for an instruction sequence.[2] Common instruction scheduling algorithms build a dependence dag from the original instruction sequence according to the constraints, and then topologically sort the dag, extracting nodes beggining with roots and ending with leaves. Thus, they choose the output instructions in order, and the resulting sequence respects the scheduling constraints. The choice of a good sequence depends on a series of decisions taken at each step, since there are usually many instructions that can be scheduled next.

The cost of a sequence depends on the sequence as a whole; when machines have deep pipelines, placement of instructions that are very far apart in the sequence can affect total cost. However, a scheduler cannot afford to examine (all) sequences to such a depth. Therefore, it must base its local choice on local information, using only the properties of the instructions currently available, or perhaps with limited lookahead. Furthermore, a scheduler is not normally allowed to backtrack. (These limitations keep the cost of basic block scheduling almost linear.)

The task of a heuristic scheduler is thus to exploit the structure of the search space and use simple local information to make local decisions in search of the globally fastest schedule. The task of learning is to identify which local information, preferably simple to compute, is useful to the heuristic scheduler. Specifically, this information will come as numeric *features* of instructions, and of short instruction subsequences, and learning will assign weights to individual features. Once training is complete, the weighted features are used by the heuristic scheduler to select the next instruction by locally evaluating an evaluation function for each admissible instruction.

We anticipate, based on reported performance on tasks of similar complexity in game playing [LM88], that with a good choice of simple heuristics, learning can be effective. The comparison will need to be made with hand-written schedulers and, where possible, with results of optimal scheduling. However, to judge the approach successful, we will have to establish three further facts: first, that the cost of learning is acceptable; second, that the learned weights are robust with respect to different subject programs; third, that we have found a set of features that is sufficiently complete to be effective for a range of architectures.

[2] All these tools are available for the Alpha 21064 chip.

5 Schedule

We have already outlined the activities proposed in some detail. We now offer a year by year schedule of the major activities:

Year 1: We will build the tools needed for exploration of learning basic block instruction scheduling: code analysers, schedulers, simple simulators, a heuristics engine and finally a learning engine. We will conduct experiments to examine the efficacy of learning. We will limit investigation to one initial machine, namely the Alpha 21164 we are requesting, and we will compare simulation results against measurements.

Year 2: We will extend the work to other existing (at that point) architectures. We will extend the work to whole procedures. We will integrate the tools into a compiler back end. Assuming that Year 1 results are positive, we will actively seek industrial partners to continue work on the broad codesign approach.

Bibliography

[Bak91] Henry G. Baker. Precise instruction scheduling without a precise machine model. *ACM Computer Architecture News*, 19(2):4–8, December 1991.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation.*, pages 275–284, 1989.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems.*, pages 122–131, Santa Clara, California, April 1991.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. In PLDI [PLD91], pages 229–240.

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In PLDI [PLD91], pages 241–255.

[CH90] Fred Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[Cha82] G.J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction SIGPLAN Notices.*, volume 17, June 1982.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In PLDI [PLD91], pages 192–203.

[GM86] Philip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM SIGPLAN 86 Conference on Programming Language Design and Implementation.*, pages 11–16, 1986.

[GR90] M.C. Golumbic and V. Rainish. Instruction scheduling beyond basic blocks. *IBM Journal Research Development.*, 34(1):93–97, January 1990.

[IAS+] Arthur Ieumwananonthachai, Akiko N. Aizawa, Steven R. Schwartz, Benjamin W. Wah, and Jerry C. Yan. Intelligent process mapping through systematic improvement of heuristics.

[Kar84] Michael Karr. Code generation by coagulation. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction SIGPLAN Notices.*, volume 19, pages 1–12, June 1984.

[LM88] Kai-Fu Lee and Sanjoy Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36:1–25, 1988.

[Mas87] Henry Massalin. Supperoptimizer – A look at the smallest program. In Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, pages 122–126, Palo Alto, California, 1987.

[Mor91] W.G. Morris. CCG: A prototype coagulating code generator. In PLDI [PLD91], pages 45–58.

[PF91] Todd A. Proebsting and Charles N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. In PLDI [PLD91], pages 256–267.

[PLD91] *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation.*, Toronto, Ontario, Canada, June 1991.

[San90] Vatsa Santhaman. Register allocation across procedure and module boundaries. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation.*, 1990.

[War90] H.S. Waren. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal Research Development.*, 34(1):85–92, January 1990.