Machine-dependent ILP optimization capabilities

1. Framework

The framework used for machine-dependent ILP optimizations in Trimaran provides advanced capabilities and support for experimenting with innovative, forward-looking ILP architectures and the compiler modules needed to generate high-performance code for these architectures.

1.1 Internal representation

The internal representation supports region-based analyses, optimizations and transformations upon code that can be in either sequential form or dependence graph form, both of which are use a uniform graph-based representation. The sequential form is used during analysis and classical optimizations; the dependence graph form is used during height-based ILP optimizations such as critical path reduction and scheduling.

The representation supports the use of control-flow, memory dependence and cache miss profile information during compilation.

1.2 Support for EPIC architectures

The framework is designed to support the advanced features of EPIC style of architectures such as predication and speculation. It provides rich support for analyzing and transforming code that contains predication and expanded virtual registers (EVRs). Each analysis module presents the analysis results to the compiler through its own interface. The predicate query interface provides a uniform way for modules to ascertain the relationship between predicates.

1.3 Machine description

Elcor uses a Machine Description Database to represent the properties of the target HPL-PD processor. Thus, it can be re-targeted to a different HPL-PD processor merely by changing the Machine Description Database. A high-level textual language, called HMDES, is used to describe the machine database.

There is support for a broad space of realistic machines including machines with non-unit latencies and complex reservation tables. Both equals and less-than-and-equals latency models are supported.

The Machine Description (Mdes) query interface allows modules to get the requisite details about the target processor without "hard-coding" such information in the modules. The query interface supports scheduling and register allocation. It also provides some basic information about machine operations, for example, the input and output operand format of operations.

The scheduling interface provides latencies for dependence edges in a data dependence graph. It also presents a *resource usage manager* to a scheduler to support allocation and deallocation of resources to operations during scheduling.

The register allocation interface provides information about the structure of register files and their capacities to the compiler.

1.4 Textual IR interface

Rebel, the textual version of the internal representation, permits one to tap into and out of Elcor for debugging purposes and for inter-operability with other compiler infra-structures. The Rebel reader constructs internal IR from Rebel. The Rebel writer prints a Rebel file corresponding to the internal IR. Currently, certain parts of the IR are not printed; these include the *generic attributes* on regions or edges, and some of the *typed attributes* for which a read/writer module does not exist.

2. Modules

2.1 Predicate query system

The details of analyzing predicated code is encapsulated in the predicate query system (PQS). Analysis and optimization modules use PQS to perform symbolic operations on predicate expressions. Any approximation in symbolic evaluation of predicate expressions generates conservative results, resulting in conservative but correct analysis and optimization results. Whether a particular symbolic evaluation is precise or approximate depends on the PQS implementation that is used.

2.2 Data-flow analysis modules

The three data-flow analysis modules are live variable analysis, reaching definitions analysis and available expression analysis. All modules can analyze predicated code quite accurately and live variable and reaching definitions analysis also can analyze code using expanded virtual registers (EVRs) accurately. All three analysis modules compute results conservatively in analyzing code where usage of predicates or usage of EVRs is too complex.

Live variable analysis operates on a region of code and annotates liveness information on the control flow edges between the basic-blocks and hyperblocks. This module can also compute the upward exposed defined variables, downward exposed used variables, and downward exposed defined variables.

Reaching definitions analysis computes def-use chains for a region of code and annotates the information on the region. Available expression analysis results are also annotated on the region being analyzed, and expression availability can be queried at any point in the control flow graph of this region.

2.3 Control-flow analysis modules

Control flow analysis modules operate on a control =-flow graph where all nodes of the graph are basic-blocks. The three control-flow analysis modules are dominator/post-dominator analysis, control dependence analysis, and loop detection. Each analysis module has a data structure that encodes the analysis results. Loop detection module also identifies basic induction variables in the loops.

2.4 Control-flow transformations

Two control-flow transformations that are implemented in Elcor, tail duplication and if-conversion, are useful building blocks for more complex transformations. Tail duplication is useful in region formation, particularly in eliminating entry points from multiple entry regions. If-conversion is useful in converting single entry multiple exit regions into hyperblocks.

2.5 Optimizations

Elcor provides a set of classical optimizations that treat predicated code and code containing EVR's. These optimizations are

- dead code elimination,
- local copy propagation (forward and backward),
- global forward copy propagation
- common sub-expression elimination
- loop invariant code removal

Two other optimizations are used to enhance the instruction level parallelism within a program. These are predicate speculation, which allows operation speculation within if-converted code, and global register renaming, which removes output and anti-dependences.

2.6 Control-height reduction

In branch -intensive programs, performance is often limited by branch dependences and branch throughput requirements. Control-height reduction is a technique which transforms programs to reduce the performance limiting effects of branch dependences and the number of branches executed. These modules analyze hyperblocks and selectively apply control-height reduction to improve ILP performance.

2.7 Dependence graph construction

Dependence graph construction is a scheduling-model specific phase of Elcor that introduces dependence edges to an operation graph. The edges include flow anti and output dependences for registers, memory dependence edges and control dependence edges that restrict code motion across branch operations.

Dependence graph form of a region can be used both for scheduling and for guiding optimizations that consider critical path lengths in a program.

2.8 Modulo scheduling of counted loops

Loop scheduler consists of two parts. Modulo scheduler allocates resources for the loop kernel subject to an initiation interval. Stage scheduler moves operations across stages in order to reduce register usage of the loop. The existing implementation of loop scheduling generates *kernel only* code, which requires support for predication and rotating register files.

2.9 Acyclic Scheduling of superblocks/hyperblocks

There are three variations of acyclic scheduling in Elcor. Cycle scheduler, backtracking scheduler and meld scheduler.

Cycle scheduler generates a schedule by constructing instructions from operations for each issue cycle in order.

Backtracking scheduler is a modified version of the cycle scheduler which can either do limited backtracking to only support scheduling of branch operations with branch delay slots, or do unlimited backtracking.

Meld scheduler is a modified version of the cycle scheduler that can propagate operation latency constraints across scheduling region boundaries. This results in tighter scheduling of operations across region boundaries.

Acyclic scheduling is performed twice on a program, once before register allocation and once after register allocation.

2.10 Rotating register allocator

When a counted loop is software pipelined, a set of virtual registers in the loop are designated as rotating registers. Rotating register allocator allocates such registers to the rotating register files right after modulo scheduling. Stage scheduling can be used after modulo scheduling to reduce the rotating register requirements of a loop. The remaining registers are allocated to the static register file after scalar scheduling of the rest of the program.

2.11 Lcode reader/writer

Elcor also has a Lcode reader/writer that constructs Elcor IR from Lcode and prints Lcode. The Lcode interface uses Impact Lcode parser and printer to construct the Impact IR, and performs internal IR-to-IR translation between Impact and Elcor IRs. Not all constructs of Elcor IR can be represented in the Impact IR therefore generation of Lcode from Elcor may not be possible at all points of compilation.