Robert Giegerich, Susan L. Graham (editors):

Code Generation - Concepts, Tools, Techniques

Dagstuhl-Seminar-Report; 13
20.-24.5.1991 (9121)

ISSN 0940-1121

Tel.: +49-6871 - 2458
Fax: +49-6871 - 5942

Das Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI) ist eine gemeinnützige GmbH. Sie veranstaltet regelmäßig wissenschaftliche Seminare, welche nach Antrag der Tagungsleiter und Begutachtung durch das wissenschaftliche Direktorium mit persönlich eingeladenen Gästen durchgeführt werden.

Bezugsadresse: Geschäftsstelle Schloß Dagstuhl
Informatik, Bau 36
Universität des Saarlandes
W - 6600 Saarbrücken
Germany
Tel.: +49 -681 - 302 - 4396
Fax: +49 -681 - 302 - 4397
e-mail: office@dag.uni-sb.de

Workshop on Code Generation

Schloß Dagstuhl, May 20-24 1991

organised by

R. Giegerich and S. L. Graham

Summary

The goal of the workshop was to evaluate the state of the art and to point out the major directions of research in code generation for the coming years. Most of the contributions were arranged around four topics:

- Tools and techniques for code selection
- Code generation for parallel architectures
- Register allocation and phase ordering problems
- Formal models and validation

Besides these, there were contributions on some new topics such as dynamic compilation or object-oriented methods for code generation.

The workshop had 36 participants, 20 from Europe and 16 from the USA.

Tools and techniques for code selection

It is widely agreed that tree pattern matching is the technique of choice for code selection. Much discussion was devoted to relating the BURS approach, tree parsing and the new approach of "regular controlled rewriting". While the pure matching problem seems solved satisfactorily, the big problem to be solved is the integration of the pattern driven code selector with other code generation tasks. "Considerate code selection" allows to postpone decisions in the code selection phase. By contrast, existing tools incorporate a specific way of where decisions are made.

An engaged discussion on specification techniques (a protocol of this discussion is included) lead to the decision that a group of the participants would cooperate to make a pure BURS system available for public distribution.

Code generation for parallel architectures

Coarse grain parallelism: This topic is characterised by attempts to provide a high-level, architecture independent model of parallelism to be used in programming and, on the other hand, programming techniques and languages features that expose the architecture to the programmer. The latter reduces compiler complexity, but also program portability. A summary of the discussion of this topis is included.

Fine grain parallelism: While a general model of pipeline scheduling is still not available, some comparative results for different RISCs and different scheduling techniques are now available. It is open how to characterize the class of machines for which these results are valid. As expected, interaction between scheduler and register allocator is crucial.

Register allocation and phase ordering

Register allocation interacts with all code generation tasks. Contributions treated techniques for code rearrangement, life range splitting and multi-level window models.

Formal models and validation

Formal methods try to enhance the reliability of code generators by providing a declarative meaning of code generator descriptions and formal proof techniques. A number of interesting approaches were presented. Some (parts of) simple code generators have been proved correct mechanically, some more realistic ones have been proved "by hand", but the effort is usually immense. Currently, no proof techniques are available for the code generation techniques actually used in practice. Both sides must advance - future tools should provide a declarative semantics, and validation methods must be geared to code generation techniques.

Acknowledgements

The workshop provided an excellent forum for the exchange of ideas, bringing together groups of experts that are unlikely to meet at the side of a more general conference. Strong participation from the USA was made possible by a travel grant from NSF.

The concept and implementation of the Dagstuhl Institute was accepted enthusiastically by the participants. The charming surroundings and the relaxed, but highly professional management helped to make this workshop a pleasant and rewarding experience.

Participants

Bodin, François
IRISA - Campus de Beaulieu
35042 RENNES Cedex
France
francois.bodin@irisa.fr

Bowen, Jonathan
Oxford Univ. Comp. Lab.
Programming Reseach Group
11 Keble Road
Oxford OX1 3QD
England
Jonathan.Bowen@prg.oxford.ac.uk

Boyland, John
Computer Science Division - EECS
University of California
511 Evans Hill
Berkeley, CA 94720
USA
boyland@sequoia.berkeley.edu

Bradlee, David G.
University of Washington
Dept. of Compter Science, FR -35
Seattle, WA 98195
USA
dgb@cs.washington.edu

Briggs, Preston
Rice University
Dept. of Computer Science
P.O. Box 1892
Houston, Texas, 77251-1892
USA
preston@titan.rice.edu

Buth, Bettina
Institut für Informatik
Universität Kiel
Preußerstr. 1 - 9
2300 Kiel 1
Germany
bb@causun.uucp

Emmelmann, Helmut
GMD an der Universität
Karlsruhe
Haid-und-Neu-Str. 7
7500 Karlsruhe
Germany
emmel@karlsruhe.gmd.dbp.de

Fédèle, Carine
I3S-CNRS
Université de Nice
250, Ave. A. Einstein
06560 Sophia Antipolis
France
carine@cerisi.cerisi.fr

Ferdinand, Christian
Universität Saarbrücken
FB Informatik
Im Stadtwald 15
6600 Saarbrücken 11
Germany
ferdi@cs.uni-sb.de

Fisher, Josh
Hewlett-Packard Labs
30 Griggs Road
Brookline, MA 02146
USA
jfisher@hplabsz.hpl.hp.com

Fraser, Christopher W.
AT&T Bell Labs
600 Mountain Ave 2C-464
Murray Hill, N.J. 07974
USA
cwf@research.att.com

Freericks, Markus
TU Berlin
1000 Berlin
Germany
mfx@mozart.cs.tu-berlin.de

Freudenberger, Stefan
Apollo Systems Div.
Hewlett Packard
300 Apollo Drive
Chelmsford, MA 01824
USA
stefan@apollo.hp.com

Ganapathi, Mahadevan
Computer Science Division
University of California Davis
Davis, CA 95616
USA
ganapath@cayenne.eecs.ucdavis.edu

Giegerich, Robert
Universität Bielefeld
Technische Fakultät
P.O.Box 8640
4800 Bielefeld 1
Germany
robert@techfak.uni-bielefeld.de

Graham, Susan L.
Computer Science Division - EECS
University of California
511 Evans Hill
Berkeley, CA 94720
USA
graham@sequoia.berkeley.edu

Hatcher, Phil
Dept. of Computer Science
Univ. of New Hampshire
Kingsbury Hall
Durham, NH 03824
USA
pjh@cs.unh.edu

Henry, Robert R.
University of Washington
CS & E Dept., FR-35
Seattle, WA 98195
USA
rrh@tera.com

Hogenkamp, Horst
Universität Bielefeld
Technische Fakultät
Postfach 86 40
4800 Bielefeld 1
Germany
horst@techfak.uni-bielefeld.de

Kastens, Uwe
Universität-GH Paderborn
FB Informatik
4790 Paderborn
Germany
uwe@uni-paderborn.de

Keßler, Christoph W.
Lehrstuhl für Rechnerarchitektur
FB 14 Informatik der
Universität des Saarlandes
Im Stadtwald 15, Geb. 36, Zi. 114
6600 Saarbrücken
Germany
kessler@cs.uni-sb.de

Knobe, Kathleen
Compass Inc.
550 Edgewater Drive
Wakefield, MA 01880
USA
knobe@compass.com

Kroha, Petr
Fachbereich Informatik
Fachhochschule Dortmund
Postach 10 50 18
Germany
kroha@zuse.informatik.uni-dortmund.de

Krzyzanowski, Paul
Paul Krzyzanowski
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974
USA
paul@research.att.com

Lake, Mike
Dept. of Computer Science
2413 Digital Computer Lab.
1304 West Springfield Ave.
Urbana IL 61801
USA
jmlake@kant.cogsci.uiuc.edu

Mazaud, Monique
INRIA-Rocquencourt
Domaine de Voluceau
B.P. 105 - 78153
Le Chesnay Cedex
France
mazaud@minos.inria.fr

Osborne, Hugh
Department of Informatics
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
Netherlands
hugh@cs.kun.nl

Pfahler, Peter
Universität Paderborn
Warburger Str. 100
P.O.Box 16 21
4790 Paderborn
Germany
peter@donar.uni-paderborn.de

Philippsen, Michael
Fakultät für Informatik
Universität Karlsruhe
Am Zirkel 2
7500 Karlsruhe
Germany
phlipp@ira.uka.de

Tichy, Walter F.
Fakultät für Informatik
Universität Karlsruhe
Am Zirkel 2
7500 Karlsruhe
Germany
tichy@ira.uka.de

Veldhuijzen van Zanten, Gert E.
Faculteit Informatica INF/TO H303
Universiteit Twente
P. O. Box 217
7500 AE Enschede
Netherlands
veldhvz@cs.utwente.nl

Waite, William M.
Department of
Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309-0425
USA
waite@boulder.colorado.edu

Wall, David
DEC - Western Research Lab
100 Hamilton Ave
Palo Alto, CA 94301
USA
wall@decwrl.dec.com

Wilhelm, Reinhard
Universität Saarbrücken
FB Informatik
Im Stadtwald 15
6600 Saarbrücken 11
Germany
wilhelm@cs.uni-sb.de

Wolczko, Mario
Dept. of Comp. Science
University of Manchester
Manchester M13 9PL
England
mario@cs.man.ac.uk

Program and abstracts

Monday, 20. May 1991

Morning session

◇ **Helmut Emmelmann** — "CODE SELECTION BY REGULAR CONTROLLED TERM REWRITING"

In my presentation I will introduce a new method to evaluate systems of term rewriting rules or term equations. It is called regular controlled term rewriting. The basic idea is to control the rule applications by using a regular tree automaton.

Using those systems makes it possible to describe code selectors by very high level specifications: The mapping from the intermediate code to the target code can be described by term rewriting rules or by equations in a compact way. A tree grammar with associated cost values is used to specify the set of target terms and their costs. Code selection is now the problem of rewriting a given intermediate code term into a target term of minimal cost. The description method, as far as only code selection is concerned, is similar to the work of R. Giegerich [Gie90a, Gie90b]. He uses an order sorted type system to specify the set of target terms, which is quite similar to the tree grammars we use. Because of the bigger class of term rewriting systems we can process our descriptions can be more compact and elegant. However in contrast to Giegerich we have not considered register allocation yet.

Our generation algorithm takes a description and produces a tree transducer which constructs for each input term a cost minimal target term. This tree transducer can be implemented very efficiently. We can handle a large class of term rewriting systems. For example it is possible to write equations which we consider as two rewriting rules one in each direction or even rules with only a variable on the left hand side. The term rewriting systems need to be neither confluent nor terminating. Similar to the approach in [EPL88] the transformation process is required to proceed bottom up with finite exceptions. However our method can also process term rewriting systems which do not obey the BURS-condition.

The paper consists of three parts. First it will be shown how code selectors can be specified using our method. The main part of the paper describes the algorithm which generates the tree transducer. Finally we will present experiences with a first prototype implementing our method.

◇ **Christian Ferdinand** — "TREEPARSING - EXPERIENCES MADE IN SAARBRÜCKEN"

The codegenerator tool developed at the Universitaet des Saarlandes has as input an annotated regular tree grammar (also called "machine grammar"), whose rules describe the effects (of parts) of the machine instructions as terms in an intermediate representation. These rules are regarded as reduction rules. A reduction sequence to a nonterminal of an expression given as a tree in the intermediate representation (equivalently a derivation of the tree according to the tree grammar) corresponds to one possible instruction sequence. A tree parser for the "machine grammars" is used

to examine all possible derivations and the cost annotations are used to determine a sequence with minimal costs.

The codegenerator tool was planned as an extension to the OPTRAN system. OPTRAN is a system to support transformations of attributed trees. The syntactic part of the application condition of a transformation is specified by a "tree pattern" in the sense of [HO82]. For these patterns the OPTRAN system includes an incremental pattern matcher generator, which generates bottom-up tree automata [Mön80]. These automata are represented by sets of "horizontal tree automata" [Kro75], which can usually be stored more space efficiently than a direct representation as matrices.

For a given "machine grammar", the codegenerator tool first generates a pattern matcher for the rules. The nonterminals are treated as terminal symbols. In a second phase, this pattern matcher is transformed into a bottom-up parser for the tree grammar by "simulating" derivation steps (or equivalently reduction steps) [WW88]. The representation of the parsers as "horizontal tree automata" leads to acceptable automata sizes. Using table compression methods as described in [BMW88], the parser can be stored very space efficient (e.g. approximately 7000 entries for a NSC32000).

◇ Robert Giegerich — "Considerate Code Selection"

Considerate Code Selection seperates analysis and program transformation used in code generation from actual selection, i.e. making decisions between alternative encodings of a program. In the extreme, all alternatives are fully analysed and the choice is made as the final step. Since the number of alternatives is exponential in the size of the input program, a new form of sharing is required to make this approach feasible.

Afternoon session

◇ Jonathan Bowen — "From Programs to Object Code and back again using Logic Programming"

A compiler may be specified by a description of how each construct of the source language is translated into a sequence of object code instructions. It is possible to produce a compiler prototype almost directly from this specification in the form of a logic program. This defines a relation between allowed high-level and low-level program constructs. Normally a high-level program is supplied as input to a compiler and object code is returned. Because of the declarative nature of a logic program, there is no reason in theory why object code should not be supplied and the allowed high-level programs returned, resulting in a decompiler. This paper discusses the problems of adopting such an approach in practice. A simple compiler and decompiler are presented in full as an example in the logic programming language Prolog, together with some sample output. Finally, the possible benefits of using *constraint logic programming* are considered. It is possible that the results presented could be developed to be of practical use for reverse engineering in the software maintenance process.

◇ C. W. Keßler, W. J. Paul and T. Rauber — "Register Allocation and Code Optimization for Vector Basic Blocks on Vector Processors"

We present a randomized heuristic algorithm to generate continuous evaluations for expression DAGs with nearly minimal register need. The heuristic may be used

to reorder the statements in a basic block before applying a global register allocation scheme like graph coloring. Experiments have shown that the new heuristic produces results about 30% better on the average than without reordering.

Basic blocks of vector instructions lead to vector DAGs. For the special class of quasiscalar DAGS, the problem can be reduced to the scalar case handled above provided that some machine constraints such as buffer size and pipeline depth are taken into consideration. Theorectical considerations show that there exists an interesting tradeoff-effect between strip miningan vector register spilling. Therefore we give an algorithm which computes the best ratio of spill rate to strip length with respect to the run time on the target vector processor which is given by some architecture parameters. This algorithm is suited for vector processors containing a buffer (register file) which may be partitioned arbitrarily by the user.

◇ **Petr Kroha** — "CODE GENERATION FOR A SINGLE-INSTRUCTION MACHINE"

Most contributions on this seminar describe sophisticated solutions of the code selection problem. In my contribution I describe how to avoid this problem. The Single-Instruction Computer (SIC) machine will be described which uses only one instruction in the set of machine instructions, i.e. the operation code will be omitted. The only operation is MOVE. The main processor (a Central Move Unit (CMU)) of the SIC machine only moves operands to specialized one-operation-coprocessors (Arithmetic Move Unit (AMU)). In this paper we discuss usage of AMU's which have the execution time larger than the MOVE operation has. This extension offers parallel programming of such a machine. Considerable thought has been devoted to problems of code generation with a particular regard to scheduling used in a compiler for such a machine.

Tuesday, 21. May 1991

Morning session

◇ **Michael Philippsen & Walter F. Tichy** — "COMPILING FOR MASSIVELY PARALLEL MACHINES"

This article discusse techniques for compiling high-level, explicitly-parallel languages for massively parallel machines.

We present mechanisms for translating asynchronous as well as synchronous parallelism for both SIMD and MIMD machines. We show how the parallelism specified in a program is mapped onto the available processors and discuss an effective optimization that eliminates redundant synchronization points. Approaches for improving scheduling, load balancing, and co-location of data and processes are also presented. We conclude with important architectural principles required of parallel computers to support efficient, compiled programs.

Our discussion is based on the language Modula-2*, an extension of Modula-2 for writing highly parallel programs in a machine-independent, problem-oriented way. The novel attributes of Modula-2* are that programs are independent of the number of processors, independent of whether memory is shared or distributed, and independent of the control mode (SIMD or MIMD) of a parallel machine. Similar extensions could easily be included in other languages.

◇ **Phil Hatcher** — "Compiling Data-Parallel Programs for MIMD Architectures"

We are convinced that the combination of data-parallel languages and MIMD hardware can make an important contribution to high-speed computing. We describe a compiler that translates a data-parallel variant of C to code suitable for execution on hypercube multicomputers. Dataparallel C provides a model that includes virtual processors, synchronous execution, and a global name-space. The hypercube compiler must implement these features on distributed-memory machines whose processors are running asynchronously. We present the results of evaluating the compiler using a suite of benchmark programs.

◇ **Kathleen Knobe** — "Issues in Generating Code for Distributed Memory Architectures"

Compass has built a number of Fortran-90 compilers for single instruction stream multiple data stream (SIMD) targets, including ones for Thinking Machines Corp.'s Connection Machine, MasPar's MP-1, and David Sarnoff Research Lab's Princeton Engine. In this context, we have developed some new compilation strategies for the middle and back end to optimize various aspects of the resulting code. Since communication among processors in distributed memory systems is far more costly than computations within processors, these new strategies are largely directed at either creating or exploiting locality within processors. Since most of these techniques are directed at optimizing for distributed memory and are not SIMD specific, we are currently extending the results to MIMD targets.

Here we describe three new analyses. The first, called data optimization, is designed to create locality of reference within processors. Data optimization analyzes the usage of array sections in the source and determines the relative layout of arrays with respect to each other in order to minimize interprocessor communication requirements. This phase focuses on the source usage and ignores limitations imposed by the target architecture. The second analysis, mapping, maps the arrays onto the finite number of processors in the target architecture, maintaining the relative alignments produced by data optimization. Mapping may result in multiple elements of an array on each processor. The third analysis, divide-into-regions, analyses the layout of each operand in an expression tree and optimizes the strip loops to optimize register usage. Although we also perform the standard strip loop optimizations, the issues addressed by division-into-regions are specific to SIMD architectures and are not described elsewhere.

Afternoon session

◇ **A. Asthana, H. V. Jagadish, P. Krzyzanowski** — "The Design of a Back-end Object Management System"

We describe the architecture and design of a back-end object manager, designed as an "active memory" system on a plug- in board for a standard workstation (or personal computer). We show how, with minimal modification to existing code, it is possible to achieve significant performance improvement for the execution of data-intensive methods on objects, simply by using our back-end object manager.

◇ **Mario Wolczko** — "Issues in Code Generation for Smalltalk-80 on an Object-Oriented Architecture"

The Mushroom Project at the University of Manchester has designed and is implementing a novel architecture to support object-oriented languages such as Smalltalk-80. The architecture has been designed with current compiler technology in mind, so that much of the burden of achieving good performance rests with the compiler.

This talk will outline the Mushroom architecture and compiler structure, and concentrate on issues peculiar to the architecture and source language which have a significant impact on code generation.

Some of these issues are:

- **Dynamically bound procedure invocation**

  The predominant control structure used in Smalltalk-80 is based around message sending (dynamically-bound procedure invocation). Apart from the overhead of this mechanism, it presents a number of barriers to further optimisation. The detection of cases which can be bound at compile time opens the door to significant performance gains.

- **Optimisation and efficient execution of Smalltalk blocks (similar to Lambda-expressions in Scheme)**

  Such expressions are used extensively in Smalltalk for the construction of control structures. The contexts (activation records) of all control structures are, in the general case, available as first-class heap-allocated, garbage-collected objects. Significant performance gains are to be had by detecting special cases which do not require the creation of full objects.

- **Interactive use**

  The final compiler must still preserve the interactive nature of the system (each compilation must not take more than a few seconds, and should usually be under a second), and respect the demands of the source-level debugger.

- **Register allocation and instruction scheduling**

  Because basic blocks are on the average very short (due to the frequent use of message sends), it is important to squeeze the most out of every cycle by avoid pipeline bottlenecks and collisions, and filling delayed branch slots as much as possible.

◇ **Josh Fisher** — "Instruction-level Parallelism & Speculative Execution"

This talk addresses compiling methods for instruction-level parallel processors, such as superscalars and VLIWs, with particular emphasis on techniques for speculative execution.

An operation is speculative when it is executed ahead of a conditional jump that might have prevented its execution. Although researchers have had difficulty accepting this conclusion, experiments done over the past 20 years have consistently shown the same thing:

- There is a lot of instruction-level parallelism available in most types of real programs (anywhere from a potential speedup factor of 5 to speedups limited only by the size of the data).

- If you don't do a lot of speculative execution, you can only get a little of it (perhaps a factor of 2-3).

Trying to do a lot of speculative execution led 13 years ago to a compiler technique called "*trace scheduling*". Trace scheduling considers very large windows, sometimes containing thousands of operations. The alternative, considering blocks of straight line code, leads to the binding all of the nonspeculative operations first, and the generation of very poor code.

Trace scheduling's windows are linear execution paths through the code, but obviously one can't consider all paths. Instead, trace scheduling compromises by passing conditional jumps only in the statically predicted more likely direction. This compromise works well for codes with highly predictable control flow, including many important numeric applications. But recent hard evidence has verified the intuitive feeling that this throws out too much opportunity in many general-purpose codes. In this talk, I will briefly review trace scheduling and the recent experimental data which leads one to want to extend it. I will then survey techniques which extend trace scheduling to more general code by considering operations from both sides of conditional jumps. Some of these techniques date from the original formulation of trace scheduling and soon thereafter, while others are new. Finally, I will touch on the interesting systems effects of trying to do hundreds of operations you aren't necessarily supposed to have done.

Discussion on "Parallelism"

Led by Walter F. Tichy with the aid of a few prepared slides
Summarized by Michael Philippsen

Parallel Machine Architecture and Parallel Programming

Today's sequential processors are not designed in a vacuum. Instead, they are built to fulfill the needs of extensive sets of benchmark programs and at the same time take into account the capabilities of compilers. A comparable level of maturity has not been reached in the design of parallel computers. The semantic gap between parallel hardware and high-level, parallel languages is substantial at present, and far too large to be bridged effectively by a compiler. The result is that programmers must code at a low, machine-oriented level and that parallel programs are largely non-portable. This poor state of affairs is not surprising, given that many of the variables involved in parallel system design are unknown and in a state of flux. These variables include the capabilities that parallel machines can offer, the translation and optimization techniques of compilers for parallel machines, and the appropriate high-level constructs in parallel programming languages. In addition, parallel system architecture allows many more degrees of freedom than sequential systems. In the long run, however, the practice of rewriting parallel programs for every new machine architecture is economically intolerable. A major challenge is hence the harmonization of parallel machine architectures, compilers, and programming languages, with the goal of allowing programs to be written in high-level, problem-oriented languages, while developing compilers that translate the programs into efficient target code for a wide variety of parallel architectures. Success will be measured by how well real, machine-independent application programs will

execute on real, parallel computers. Since highly parallel machines with thousands and tens of thousands of processors are already being manufactured and used commercially, this challenge requires a solution urgently.

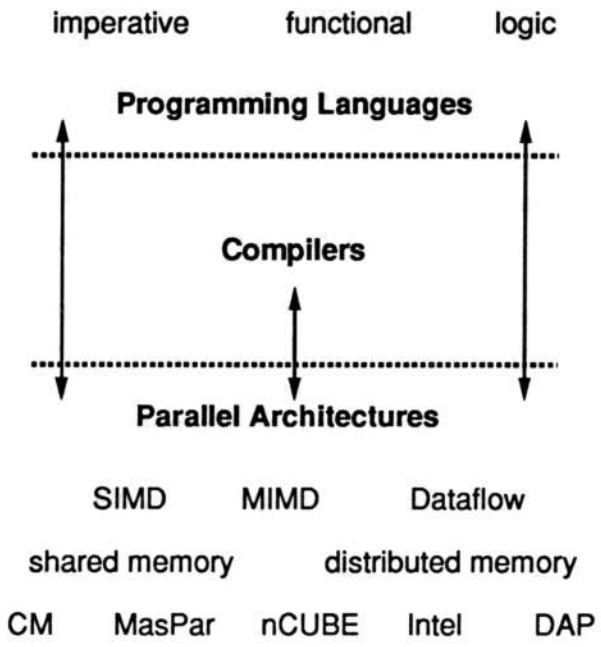

Figure 1: Interaction between Languages, Compilers, Architectures

The purpose of the discussion was to approach this challenge from the language and compiler designer's viewpoint. The questions put to the audience were as follows:

- What are the needs of the programmers?
- What features of parallel computers are germane and must be reflected in the programming languages?
- What features of parallel computers are accidental or irrelevant for machine-independent programs?
- What are the major questions faced by compiler writers?
- What properties should parallel hardware possess to allow for efficient, compiled programs?

A view shared by many in the audience was that the approach of automatically parallelizing existing, sequential code should not be followed intensively. Although there is overwhelming economic justification for this approach, it will meet with only limited success in the short to medium term. The goal of automatically producing parallel programs can only, if ever, be achieved by program transformation systems that start with problem specifications and not with sequential implementations. In a sequential program, too many opportunities for parallelism have been hidden or eliminated.

A long term goal could be to develop *interactive* program transformation systems that assist programmers in parallelizing programs and provide feedback and guidance. The problem with this idea is that the approach of semi-automatic program transformations is still an object of active research, even for sequential programs. In the medium term, a production-quality transformer for deriving realistic, parallel programs

is unlikely to appear. The traditional method of teaching algorithms and formulating them in programming languages with explicit parallelism is likely to be more successful, especially since the body of known parallel algorithms is large and growing rapidly. Initial indications seem to be that writing parallel software is not significantly harder than writing sequential software, provided the languages and support tools are adequate.

Concerns of the Programmer

The following is a general list of requirements on programming languages and support tools. The requirements are not new; they are borrowed from the world of sequential programs, but apply equally well to parallel programs.

- The programming language should permit clear expression of algorithms and systems architecture, to ease the writing, reading, verification, understanding, modification, and reuse of software systems.
- Programs should be portable to a wide range of hardware architectures.
- Programs should run (after compilation) with satisfactory efficiency and with effective utilization of the available hardware resources.
- Support tools for debugging, testing, and measuring of parallel programs should be available.

Current practice is quite different: Existing parallel programming languages do not allow clear expression of algorithms, parallel programs are not portable, and support tools are often poor. The goal of satisfactory efficiency can often be met only by writing low-level, machine-dependent code.

But what are properties of parallel architectures that should be visible in parallel programs? Should the number of processors be visible? Is it necessary for the programmer to know about the organization of memory or the layout of the data? Are explicit communication protocols necessary, or will the compiler be able to insert them into the generated code by analyzing the data usage patterns? Should the control mode be visible, i.e., should there be different programming languages specialized for SIMD, MIMD, Dataflow or systolic computers?

PRAM. The PRAM is clearly the best studied approach to parallel programming today. Most algorithms are formulated for the PRAM model. The shared memory of this model makes programs much easier to understand than for models with distributed memory and explicit message passing.

During the discussion we could not agree on the performance losses involved in mapping the abstract PRAM model onto a real machine. If this mapping is possible without adding asymptotic complexity, the PRAM is likely to be broadly accepted. But this will be difficult to achieve, since the PRAM completely ignores the memory hierarchies of today's machines.

Due to the lack of high-level parallel languages, an adequate approach to parallel programming is still to develop a PRAM algorithm first and then to translate it (by hand or compiler) to fit the target machine.

Modula-2*. One approach to the above questions is Modula-2*, which is presented in a separate article in this volume. Modula-2*, an extension of Modula-2, is problem-oriented in the sense that the programmer can choose and mix the degree of parallelism, i.e., the number of processors, and the control mode (SIMD-like or MIMD-like), as needed by the intended algorithm. An interconnection network is not directly visible

in the language. A shared address space among all processors is assumed, though not the necessarily shared memory. There are no explicit message passing instructions; instead, reading and writing locations in shared address space subsume message passing. Special data allocation constructs help control the access times in distributed memory.

On the features of parallel machines that should be visible in programming languages we noted the following. The number of processors needs to be available (as a constant or variable), since algorithms that adapt to the number of processors often perform superior compared to those that don't. However, there are many algorithms for which simple, system-provided multiplexing of processors ("processor virtualization") is sufficient. In those cases, programmers should not be forced to program virtualization explicitly.

We do not think that programmers should be forced to specify much detail concerning data layout and network structure, since both of these tend to be major sources of machine dependence. Instead, we should develop tools that automatically or semi-automatically embed the data and access patterns into given hardware structures. Explicit message passing should also be avoided in high-level programming languages, because it is tedious and error-prone to program massively parallel message passing, and also because message passing instructions are intimately tied to the way a problem has been mapped onto a given architecture. Thus, message passing tends to cause machine dependence that is difficult to remove and scale.

Whereas in Modula-2* the organization of the memory is invisible, Knobe spoke out in defense of the visibility of local address spaces. If locality is present at the language level, it simplifies dependence analysis and enhances the efficiency of the generated code.

Recommendations for Parallel Architectures

The last part of the discussion centered on recommendations for parallel hardware. The starting point was a list of recommendations which were collected during the work on the compiler for Modula-2* by Philippsen and Tichy.

- Hardware support for fast process creation, synchronization, and context switching.
- Shared address space.[1] All processors should be able to generate addresses for the entire memory on the system. Even the front-end's memory should be part of that address space.

  For pointers, system wide addresses are important, because otherwise they would have to be simulated quite inefficiently in software.
- Uniform memory access instructions. Most parallel machines today provide a set of instructions for accessing local memory, a second one for accessing memory in neighbors, and a third set for accessing distant memory units. The differences in speed are significant and therefore require that the compiler detects the faster cases. However, it is often impossible to know statically for which case to optimize. For instance, we found that in many cases it was impossible to determine in the compiler whether a procedure would access local or non-local memory. The generated code thus has to check all three cases at run-time. Such a simple and frequently repeated case analysis could be done much more efficiently in hardware.

[1] A shared address space does not imply shared memory.

- Simulating shared memory. A shared memory in which all memory units can be accessed in the same time would simplify programming and optimizing compilers greatly. Latency hiding and randomization technics could help achieve a reasonable approximation of true shared memory. Latency hiding means that each processor can initiate several memory references before receiving a response. Thus, the network serves each processor's request in a pipelined fashion. The total network bandwidth of the network must be high enough to accept and serve memory accesses for all processors at rate that is comparable with accesses to local memory.
- Autonomous addressing capability. An autonomous addressing capability means that each processor can generate its own addresses for accessing memory. The Connection Machine, for example, does not have such a facility – each processor must use the same address into its own, local memory for each parallel instruction. The lack of autonomous addressing not only makes many applications awkward to write, especially if they use pointers, but also precludes certain optimizations in processor virtualization.
- Single instruction set. SIMD machines today typically have different instruction sets for front-end and parallel processors. This property implies that the code generator of the compiler has to be written twice. Also, each procedure has to be translated twice, doubling code size. A single instruction set would simplify this aspect.
- Small instruction set. The Connection Machine, for example, offers about 400 instructions. As in sequential compilers, only a few dozens of these instructions can actually be generated. Clearly, a detailed study determining the most frequently used instructions in parallel programs is desperately needed.

Wednesday, 22. May 1991

Morning session

◇ David G. Bradlee — "Retargetable Instruction Scheduling for Pipelined Processors"

Retargetable code generators developed in the last decade have focused on instruction selection for complex instruction set computers (CISCs). These code generators could ignore instruction scheduling and, in many cases, did not perform global register allocation. For reduced instruction set computers (RISCs), however, the compiler's emphasis must be shifted to instruction scheduling. In addition, register allocation is at least, if not more, important for RISCs, because they typically have more registers and all computation requires registers.

Retargetability remains important for RISCs, but the retargetability issues are different than for CISCs. First, the machine specification must capture most scheduling information. Second, the interaction between register allocation and instruction scheduling is significant, because the scheduler needs registers to enable operations to be overlapped.

My study of retargetable instruction scheduling for RISCs comprises three components: the design and implementation of the Marion Code Generator Construction System; the creation and analysis of code generation strategies; and the investigation of the interaction between strategies and architectural features.

Marion contains a code generator generator that inputs a machine description and outputs target-dependent data. The machine description contains most of the information necessary to produce efficient code for RISCs, including instruction scheduling requirements. Code generators have been built for the MIPS R2000, Motorola 88000 and Intel i860, along with a number of variations on these architectures.

The code generation strategy refers to the invocation order of and level of communication between instruction scheduling and register allocation. I examine three strategies, including one I developed, RASE, that integrates instruction scheduling and register allocation. On a computation-intensive workload for three RISCs, RASE produces significantly better code than the Postpass strategy, which does not integrate the two phases, and slightly better code than the other strategy, IPS, which integrates the two phases to a lesser degree than RASE.

In the investigation of the interaction between strategies and architecture, I vary architectural features, including register set size and structure and operation and load latencies, and examine the effect across the three code generation strategies. On a computation-intensive workload, 64 registers yields a significant improvement over 32 for Postpass, but very little improvement for IPS or RASE.

◇ John C. Ruttenberg & Stefan M. Freudenberger — "Phase Ordering of Register Allocation and Instruction Scheduling"

Register allocation and instruction scheduling are often separated due to the complexity of each. But if register allocation is performed before scheduling, it may introduce artificial data precedence, keeping the instruction scheduler from doing its best job. On the other hand, waiting until after scheduling to perform register allocation may produce impossible schedules. In this paper we present a unified approach to instruction scheduling and global (beyond basic blocks) register allocation.

We assume that we compile for a RISC machine with exposed pipelines and a large amount of instruction set parallelism that must be statically specified. Instruction scheduling is required in order to exploit any part of the performance potential of this machine; without it, performance would be very uncompetetive. We also assume that registers are often the most critical instruction scheduling resource; therefore we believe that we must delay register binding decisions until scheduling time in order to give the instruction scheduler the greatest possible freedom in picking registers.

This technique has been used in Multiflow's Trace Scheduling compilers, i.e., a commercial quality implementation of these ideas has been done that shows that this approach is both viable and effective.

◇ François Bodin, William Jalby, Christine Eisenbeis, Daniel Windheiser — "Window-Based Register Allocation"

In this paper, we consider register allocation optimization problem for loop array references as a particular case of memory hierarchy management optimization. This permits us to exploit techniques for data locality estimation and improvement usually done in the framework of cache or local memories management. First we recall the concept of a "reference window" that serves as a good tool for both data locality evaluation and management. We present a register allocation procedure based on the window concept which we show to be usable in a compiler system. Estimations of expected

speedups are done. Then we study how loop restructuring techniques (interchanging and tiling) can help improve data locality. Experimental speedup measurements validating the interest of the approach are given for two RISC processors.

Afternoon session

◇ Preston Briggs & Keith D. Cooper & Linda Torczon — "AGGRESSIVE LIVE RANGE SPLITTING"

The importance of register allocation was recognized when building the first optimizing compiler. Since that time, high-quality register allocation has remained an important consideration in the design of optimizing compilers. Currently, graph coloring allocators dominate the field.

Fabri and Chow have independently observed that splitting a single live range into several pieces and considering the new, smaller live ranges separately can produce an interference graph with lower chromaticity. Chow and Hennessy used this idea, called *live range splitting,* as the basis for a new allocator that avoided spilling when splitting was possible.

Live range splitting has several merits. If an entire live range is spilled, as in Chaitin's work, its value will reside in a register only for short periods around each definition and use. Splitting allows the value to stay in a register over longer intervals – often an entire block or over several blocks. With luck, the new live range will be large enough to extend over a complete loop.

Unfortunately, live range splitting is difficult. There are two fundamental problems: picking live ranges to split and picking places to split them. While optimal solution of either problem is surely NP-hard, we have developed a collection of heuristic techniques that extend Chaitin's allocator and cooperate to address both problems.

Aggressive live range splitting Observing that split points often become spill points, we define split points on edges entering and exiting loops with high register pressure. To avoid choosing live ranges to split, we split all live ranges at each split point.

Conservative coalescing If we allowed coalescing to run as usual, all effects of splitting would be removed. Instead, we use a limited form of coalescing to remove excess splitting where the colorability will not be adversely affected.

Biased coloring Conservative coalescing is unable to remove all excess splits without expensive examinations of the graph. However, we can achieve further coalescing during color selection by biasing the color spectrum for each node to favor colors that eliminate splits.

We have implemented these techniques in an experimental allocator and the early results are promising; however, a fair amount af additional work remains.

Thursday, 23. May 1991

Morning session

◇ David Wall — "Experience with a Software Architecture"

Inconvenient or technology-dependent features of a processor can be hidden by surrounding the actual hardware with a higher-level architecture implemented in software. This high-level "software architecture" can be used as the target of all high-level compilers and as the official assembly language for a machine or family of machines. The implementation of the software architecture controls every bit of low-level code produced. This lets it perform very global optimizations that would be impractical or unsafe in a more traditional compiler. Moreover, the software architecture can provide instrumentation services that are more flexible than those provided by hardware architectures. The Mahler software architecture for WRL's Titan family has provided a framework for exploring these possibilities. Within it, we have implemented an intermodule register allocator, a pipeline instruction scheduler, and a variety of high-level and low-level performance analysis tools.

◇ Annie Despland & Monique Mazaud — "PAGODE: automat derivation of back ends with peephole optimizers"

PAGODE is a code generator generator, based on tree rewritings. The IR to be input to the code generator is a term of an abstract data type such that the elementary instructions act on cells via operators denoting access path to cells. The target machine specification is hierarchically organized into concepts corresponding to the main features of the instruction set of the processor : locations, addressing modes, instructions. The semantics of each concept is specified by a template which is a term of the same abstract data type.

The instruction selection step applies a set of rewriting rules to the IR term. These rules are driven by tree templates derived from the target machine specification

Basically, each instruction of the IR is matched with an instruction template. In the context of such an instruction template, the operands are matched with addressing modes templates. If an operand does not match any addressing mode template and a subterm does, then the location designated by this subterm is stored in a temporary location using a universal store. The IR is rewritten using this temporary location.

The concepts of temporary location and universal store enable to separate clearly the register allocation phase from the instruction selection one. Furthermore, this allows to make a clear cut between the choice of an actual storage for a temporary and the choice of its name.

The result of the instruction selection step is a sequence of instances of instruction templates. Such a sequence uses resources which are either actual resources or temporary resources of a universal type of storage. It is necessary to bind such temporary with a set of valid actual storage types.

The register assignment is performed by graph coloring taking advantage of the results of the binding step.

Peephole optimization rules can also be applied. They are produced by the instantiation of generic rules using the target machine specification and some computations on the semantics of instructions.

◇ Ralph E. Johnson & Carl Mc Connell & J. Michael Lake — "The RTL System: A Framework for Code Optimization"

The construction of compiler front-ends is understood well enough for a great deal of the work to be automated. This paper describes type RTL System, which helps construct the rest of the compiler — the optimizer — by providing a flexible set of classes with a large number of predefined algorithms that the compiler writer can customize. It also includes a traditional table-driven code generator and peep-hole optimizer. The RTL System differs from systems to construct compiler front- and back-ends because it does not specify the optimizations with a specialized language, but is instead an object-oriented framework. This paper describes the framework and how it can be used to build a code optimizer.

◇ Mahadevan Ganapathi — "Prolog based Compiler Back-end Generation"

Prolog is used as a back-end language for the specification and implementation of optimizing code generators. It is used to reformulate pattern-matching code generators and implement retargetable compiler back-ends. A comprehensive set of optimizations is integrated into this framework and uniformly applied to produce high quality code. The more precise the optimization rules are, the better is the discrimination.

Afternoon session

◇ David Wall — "Systems for Late Code Modification"

Modification of code after it has been generated is useful for a variety of applications including some kinds of late optimization and many kinds of high-level and low-level instrumentation and simulation. Two systems that have been developed for doing this are the code modification part of my Mahler system and the "pixie" tool developed independently at Mips.

The Mahler code modifier is part of the linker, and modifies object modules as they are being linked. This has several advantages. An object file contains a relocation dictionary and loader symbol table, so Mahler can recognize address references and can correct them to reflect the changes made. The symbol table also provides a channel for additional information that the compiler can include to explain tricky things in the compiled code.

Mahler has the added advantage that the Mahler compiler produces all of the object modules: it serves both as the back end of all high-level compilers and also as the only available assembler. This means that any coding conventions followed by the Mahler compiler are guaranteed to hold throughout the entire program.

Mahler has the disadvantage that the linker is nonstandard. Moreover, a user who requests a particular application must re-link the program, and so must know what object modules and libraries make up the program.

The pixie system works differently. Pixie modifies an executable file that has already been fully linked. The relocation dictionaries are gone, and the loader symbol table may be gone as well. This means that a user can invoke pixie on an executable without knowing or caring how it was built. However, it also means that pixie must be conservative in many ways: for instance, each indirect jump in the original code is replaced by a sequence of instructions that jumps via a huge address translation

table incorporated into the modified executable. This kind of overhead makes pixie an unsuitable medium for modifications that optimize, though it is still very convenient for modifications that instrument.

I am exploring two intermediate points between Mahler and pixie.

The first is "dixie", which acts on an executable as pixie does, but assumes that it was generated using the Mips compiler conventions. Most executables include some assembly code from libraries, which might not follow these conventions, so dixie looks for library routines that it knows violate the conventions but that it can understand anyway. In many cases this allows dixie to modify a program without needing the big jump table, though programs that contain unexpected indirect procedure calls must still include the table.

The second is "epoxie", which assumes that the program has been completely linked using an incremental linker that leaves the relocation dictionaries in place. (Unix linkers normally have an option that does this.) This gives waxie some of Mahler's advantages without requiring modifications to the standard linker. A jump table is never required, and the code modification process can (I hope) be unintrusive enough to use for optimization as well as for instrumentation.

◇ **Susan Graham** — "PATTERNS, TRANSFORMATIONS AND ATTRIBUTES"

As part of our research on dynamic compilation, John Boyland and I draw on some results of Charles Farnum from his December 1990 Ph. D. dissertation, titled "*Pattern-Based Languages for Prototyping of Compiler Optimizers*". The pattern language, used for rewrite systems 'compiled' into bottom up regular tree automata, is an extension of the usual tree pattern languages. The additions include typed wildcards (where types are sets of regular tree patterns), horizontal iterations to support operations of varying arity, and vertical iterations to handle repetitive constructs such as left-associative addition trees. An attribute system is organized around the enriched pattern matching system to support factoring of the description by attributes rather than by abstract syntax rules.

◇ **John Boyland & Susan L. Graham** — "CODE GENERATION FOR DYNAMIC COMPILERS"

The design of dynamic compilers, that is, compilers that preserve execution state when newly compiled code is patched into an executing image, presents the compiler writer with a number of difficulties. First, all the complexity of a standard compiler is present. Second, enough intermediate information must be maintained to allow the compilation to proceed incrementally. Third, the incremental incorporation of newly compiled segments of code must disturb the existing execution state as little as possible. All of these factors are compounded when the desired level of granularity is small, such as at the level of statements or expressions. We describe our proposed method for automatically generating dynamic compilers and explain how the method handles these issues.

◇ **Robert R. Henry** — "SMALL, FAST AND OPTIMAL INSTRUCTION SELECTORS AND SOME SCANDALOUS POTENTIAL APPLICATIONS"

Bottom-up tree pattern matchers based on BURS theory are becoming the method of choice for instruction selection. Although building the tables from a tree grammar is both complex and expensive, the final tables are consumed by a simple and intuitively appealing interpreter invoked when traversing the tree. By folding the interpreter into the tables and carefully choosing between tables and specialized hand code, BURS tables can be encoded into a small amount of space and be used to drive a very fast pattern matcher and selector. Typical sizes are 40kBytes with a ratio of 300 instructions executed to instructions generated.

This fast code generator let us contemplate generating object code for immediate use. Self modifing code, for appropriate definitions of "self" and "modify" can now be contemplated, abstracted and efficiently implemented.

◇ **Christopher W. Fraser** — "SPECIFYING CODE GENERATORS"

This talk will describe experience with three different code generator generators and their specification languages. The first system was based on a peephole optimizer driven by a formal description of the target machine. This approach was adapted for use in the Gnu C compiler, so its practicality has been proven. More recent work shows that the specifications can be particularly succinct.

The second system is based on a language for concise expression of hand-written peephole optimizations. Specifications take 100–200 lines and compile into a fast, monolithic program that accepts dags annotated with intermediate code, and generates, optimizes, and emits code for the target machine. These code generators are used in `lcc`, a compiler for ANSI C on the VAX, Motorola 68020, SPARC, and MIPS R3000. Its local code is comparable with that from other generally available C compilers, but the compiler is much smaller and faster. `lcc` has seen production use at Princeton University and AT&T Bell Laboratories for over two years.

The third system uses BURS theory, in one of its early applications in a production compiler. The system is under development, but early experience has exposed unexpected challenges. For example, in at least one case, it has been necessary to encode hand-written peephole optimizations as BURS grammar rules; shorter, clearer encodings are available. At least one full code generator should be complete before the conference. The talk will describe the engineering required to use BURS theory in a production compiler.

The approaches have different strengths: for example, the first accepts specifications that are the closest to a pure description of the machine, the second is the most flexible, and the third is the most attractive from a formal viewpoint. The talk will present the pros and cons of the competing approaches, discuss the prospects for reducing the trade-offs between them, and argue for generating *all* code generators from "pure" machine descriptions.

Discussion on "Code Generator Specification Techniques"

Led by Chris Fraser
Summarized by John Boyland and Helmut Emmelmann

Major participants:

John Boyland, Helmut Emmelmann, Robert Giegerich, Susan Graham, Robert Henry, Uwe Kastens, Bill Waite, David Wall

Code generator specifications

The discussion started with some more questions on Chris Fraser's talk: Robert Giegerich suggested specifying code selection using a description (in the spirit of the system proposed by Helmut Emmelmann in his presentation) to separate rules about the machine description from code transformation rules. A code generator description would perhaps be split into the following parts:

- machine description
- IL description
- term rewriting rules
- optimizations

Chris Fraser felt that this may lead to overly verbose descriptions, and in particular, the IL description should not be part of code generator description. It was agreed that this topic is still research.

Chris Fraser continued the discussion by asking for suggestions for a common BURS tool. He mentioned that several research groups had already started to develop their own. In order to reduce redundant work, it would therefore be desirable to have one freely available BURS tool. Chris Fraser could develop such a tool, but then AT&T would own it. Robert Henry has a BURS tool (altogether about 25000 lines of code) but no time to adapt it for general distribution. However it is not time critical to finish the common BURS tool, because for experiments and for debugging of specifications, a implementation based on the Aho/Johnson dynamic programming algorithm (DP) can be used. Only for a production compiler would the BURS tool be necessary to make it run fast. Even with the BURS tool available, it would be desirable to continue to distribute the DP tool for debugging purposes.

Discussion then centered on defining a standard input format and in particular on the method for specifying actions and costs for each BURS rule, so that development of BURS code generators can go forward. Bill Waite observed that BURS technology could be useful for applications other than code generation, for example, operator identification; and thus the input format, and in particular the action clause, should not be code generation specific. Uwe Kastens raised the issue of specifying other types of costs, such as pipeline costs, but Susan Graham remarked that BURS can only handle integer cost values which combine additively; otherwise compile time dynamic programming becomes necessary. Rather than preclude other applications, therefore, and in order to avoid handling notational convenience (as discussed below), the group agreed on a simple low-level input format with integer costs and integer action numbers. Compiler writers would then be free to develop extended BURS (EBURS) processors that would use the low-level BURS tool to do the sophisticated work and that would implement a customized version of the input language. There was no discussion on a standard output format for the low-level BURS translator.

Extending BURS

After a short break, Chris Fraser raised the following issues he had noticed when he was writing BURS specifications:

1. how could factoring be handled in an extended BURS (EBURS) ?
2. how could DAGs be handled ?
3. how could scheduling be handled ?
4. how should one split code generator specifications into a machine description and rewrite rules ?

Factoring of BURS descriptions

Chris Fraser asked how factoring could be expressed in a EBURS–language. The following example of problem (which is part of the Chris Fraser's Vax description) shows that factoring (here, factoring on binary operators) is desirable:

```
expd: (BIN,D,xd,xd)
```

Factoring should also simplify certain recurring patterns, here demonstrated with the assignment operator:

```
stmt: (ASGN,D,inx8,expd)
stmt: (ASGN,F,inx4,expf)
stmt: (ASGN,I,inx4,expl)
stmt: (ASGN,S,inx2,expw)
stmt: (ASGN,C,inx1,expb)

similarly for ARG, LOAD, ... (in the place of ASGN)
```

The Vax description became about 40% shorter with factoring using ad-hoc regular-expression-like patterns. Robert Henry proposed using some textual macroprocessing mechanism. Chris Fraser said he would prefer something more powerful and cleaner, but will use macroprocessing if nothing else is found. Bill Waite asked if it would be enough that the system allow the programmer to specify the correspondences D/inx8/expd, F/inx4/expf, etc., to be used in rules for ASGN, ARG and LOAD. No agreement on a standard factoring method was reached.

Application of BURS to intermediate code in DAG form

The problem faced here was code generation for a DAG where shared nodes represent common subexpressions. One does not always want to allocate a register for common subexpressions: even ignoring the issues of register pressure, the code may end up longer! For example on the VAX, most addressing modes provide free computation of register + constant and immediate data (32 bits); these free quantities should not be assigned to registers. Other machines have different free operands: on MIPS, only 16 bit signed constants are free and only in certain situations (as right operands of ADD etc).

The bottom-up phase of a BURS automaton works with DAGs. The code emitter, working top-down, could count visits and for each node do one of the following things:

- generate the code as normal BURS does

- evaluate the shared subtree into a register and remember the register assigned
- not produce code for the shared subtree, but instead reuse the value stored in a register before

The second and third alternative however require that BURS has decided to place the result of the subtree into a register.

Chris Fraser identified two problems when producing code for DAGs using a BURS code selector:

- how to find out or how to specify which expressions are free (and should therefore not be placed into a register)
- how to force the code selector to put something into a register

For the first problem John Boyland proposed to just use the cost values in the description, addressing modes would have zero costs. Robert Henryremarked that we had to be careful: costs not always assigned in the right places in a description.

Then Bill Waite proposed to add new rules for free productions:

Before:

X : ⟨ free1⟩
X : ⟨ free2⟩
Y : use(X) (free)
Z : use(X) (costs)

After:

Xf :⟨ free1⟩
Xf :⟨ free2⟩
X : Xf
Y : use(Xf)
Z : use(X)

For the second problem, how to get copy into a register, Robert Henry proposed to insert a copy to a register in the DAG on the bottom-up pass, if we notice we need to put it in register. Bill Waite proposed to do it on the top-down pass (when we emit code); if a subtree doesn't have zero cost we calculate it into a register and then use the register at the node. However David Wall remarked that this would not work because a pattern may match over the DAG join.

Then Helmut Emmelmann proposed to add a DAG operator into the intermediate language and to put in rules which force the subtree below the DAG operator to be evaluated into a register. Robert Henryproposed the following rules for DAG:

reg : DAG(X) 1 ⟵ cost
Xf : DAG(Xf) 0

Finally the group came up with a better solution:

reg : DAG(reg) 0
Xf : DAG(Xf) 0

These two rules force everything below DAG into a register (first rule) unless it is free (second rule).

Problem 3 and 4 were not handled in the discussion, as they were considered still research topics.

Friday, 24. May 1991

Morning session

◇ Hugh Osborne — "UPDATE PLANS"

It is a truism that a programme is a function from machine state to machine state, composed from the functions represented by individual machine "instructions". However, specifying these instructions in a functional formalism is often unwieldy and frequently far removed from potential concrete implementations. If a prototype implementation is produced using a functional language it is usually unnacceptably inefficient.

Other abstract machine specification methods also have drawbacks. Transition systems quickly reach their limit of readability, writability and comprehensability as the structure of machine configurations becomes complex. An imperative programming style is often ad hoc and quite often contains (hidden) machine dependance. An informal description is often incomplete and hard to implement. In recent years the Bird-Meertens formalism has been gaining in popularity. Work on abstract machine description in squiggol has been done, and has indeed led to new insights into classes of abstract machines, but it is still a major step from a squiggol "programme" to a Von Neumann implementation.

Update plans, proposed as a method for low level specification, have now been developed into a high level language for specifying low level activities. They are amenable to interpretation as specifications of machine state transitions while maintaining a great deal of similarity to low level code, thus allowing for efficient hand compilation to, for example, assembler.

A compiler for a subset of update plans is currently under development. The aim is to produce a series of compilers, ranging from compilers producing innefficient code but providing a wide range of compile and run time trace and debugging facilities, to optimising compilers producing rapid prototypes. A preliminary compiler was tested on a specification of an abstract machine for a simple functional language. The resultant code was of the same order of efficiency as the Miranda system (Miranda is a trademark of Research Software Limited). A specification of a more complex abstract machine, for a logical functional language, using update plans is planned, which will then be implemented.

Update plans have also succesfully been used as a teaching aid in the undergraduate compiler construction course at the University of Nijmegen, where they are used to specify the implementation of an imperative language.

◇ Veldhuijzen van Zanten — "CODE GENERATION BASED ON A FORMAL MACHINE MODEL"

We present a formal machine model and use this to derive a code-generation technique. In the model, cells and values, which constitute the machine state, play a central role. The machine state is modified under control of a program that is a syntactic object consisting of a sequence of assignments. Each assignment assigns a value expression to a cell expression. A generic language is built around these concepts, so that we can talk about them without having to resort to some specific target-machine architecture. The denotational semantics of this generic language are described in the first part of the paper. Due to the precise formal nature of the machine model, we are able to give a precise definition of the code-generation problem. This definition demands a code generator to

- be correct,
- be complete, i.e. able to generate code for any program,
- generate efficient code, and
- be efficiently implementable.

Using this definition, we derive a code-generation algorithm. As in other techniques, instructions are modelled by templates that can be glued together in order to construct a program. In traditional techniques, data-flow dependencies are used to glue the templates together. In our model, however, the glue models the control flow. As a consequence, we can handle instructions with multiple effects in a uniform way. The data-flow dependencies, however, impose more complex conditions on the rewriting process. As template matching *and* an equivalent of register-transfer lists form the basis of the algorithm, it can be seen as a middle road between template-matching schemes as described by Giegerich and the Davidson and Fraser approach.

The algorithm is based on rewriting program terms using the following scheme. A program p is examined in order to identify a useful instruction I so that there exists a reduced program p' for which the composition of the effect $\delta(I)$ of I and p' is semantically equivalent to p. The reduced program p' can be seen as an optimized version of p, where the knowledge that I is already executed is exploited in order to reduce the cost of the program.

To exploit the inherent freedom of choice in selecting instructions and in optimizing the reduced program, we rewrite jungles instead of program terms. Jungles are data structures that incorporate part of the data flow in addition to the program structure.

◇ Bettina Buth & Karl-Heinz Buth — "An Approach to Automatic Proof Support for Code Generator Verification"

In principle, program verification is the only adequate means to ensure the correctness of software with respect to precise or formal specifications. But since realistic programs and especially code generators and other parts of compilers tend to be large and complex, some mechanical support is necessary for the verification of these programs. In this paper we present the ideas of the verification support system PAMELA that is intended for the verification of programs written in a subset of Meta IVthat are specified by pre- and postconditions. PAMELA organizes the proof for such programs and is based on a special kind of term rewriting.

◇ C.A.R. Hoare, He Jifeng, Jonathan Bowen and Paritosh Pandya — " An Algebraic Approach to Verifiable Compiling Specification and Prototyping of the ProCoS Level 0 Programming Language "

A compiler is specified by a description of how each construct of the source language is translated into a sequence of object code instructions. The meaning of the object code can be defined by an interpreter written in the source language itself. A proof that the compiler is correct must show that interpretation of the object code is at least good (for any relevant purpose) as the corresponding source program. The proof is conducted using standard techniques of data refinement. All the calculations are based on algebraic laws governing the source language. The theorems are expressed in a form close to a logic program, which may used as a compiler prototype, or a check on the

results of a particular compilation. A subset of the occam programming language and the transputer instruction set are used to illustrate the approach. An advantage of the method is that it is possible to add new programming constructs without affecting existing development work.

Afternoon session

◇ **Robert Giegerich** — "WHAT CAN WE EXPECT FROM FORMAL CODE GENERATOR SPECIFICATION AND VERIFICATION TECHNIQUES?"

One to the growing concern for hardware and software reliability, mechanical assistance in correctness proofs is required. We can distinguish two such approaches. The first approach translates the design into some well-known logic, and uses an existing theorem prover for validation. The second approach starts from problem oriented notations and concepts, trying to formalize these to obtain a problem-specific validation system. The talk sketches various aproaches of both kinds, tring to evaluate their relative strenghts and weaknesses. This is intended to spawn discussion on the viability of the formal approaches.

References

[BMW88] Jürgen Börstler, Ullrich Möncke, and Reinhard Wilhelm. Table compression for tree automata. Technischer bericht, Universität des Saarlandes, 1988.

[EPL88] Susan Graham E. Pelegri-Llopart. Optimal code generation for expression trees: An application of burs theory. In *Proceedings of the 15th Symposium on Principles of Programming Languages*, pages 294–308, 1988.

[Gie90a] Robert Giegerich. Code selection by inversion of order-sorted derivors. *Theoretical Computer Science*, (73):177–211, 1990.

[Gie90b] Robert Giegerich. On the structure of verifiable code generator specifications. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–8, 1990.

[HO82] Hoffmann and O'Donnel. Pattern matching in trees. *Journal of the Association for Computing Machinery*, 29(1):69–95, January 1982.

[Kro75] H. Kron. *Tree templates and subtree transformational grammars.* PhD thesis, University of California, 1975.

[Mön80] Ullrich Möncke. An incremental and decremental generator for tree analyser. Bericht A80/3, Universität des Saarlandes, FB10, 1980.

[WW88] Weisgerber and Wilhelm. Two tree pattern matchers for code selection. In *Lecture Notes in Computer Science*, volume 371, pages 215–229, 1988.

Bisher erschienene und geplante Titel:

W. Gentzsch, W.J. Paul (editors):
Architecture and Performance, Dagstuhl-Seminar-Report; 1, 18.-20.6.1990; (9025)

K. Harbusch, W. Wahlster (editors):
Tree Adjoining Grammars, 1st. International Worshop on TAGs: Formal Theory and Application, Dagstuhl-Seminar-Report; 2, 15.-17.8.1990 (9033)

Ch. Hankin, R. Wilhelm (editors):
Functional Languages: Optimization for Parallelism, Dagstuhl-Seminar-Report; 3, 3.-7.9.1990 (9036)

H. Alt, E. Welzl (editors):
Algorithmic Geometry, Dagstuhl-Seminar-Report; 4, 8.-12.10.1990 (9041)

J. Berstel , J.E. Pin, W. Thomas (editors):
Automata Theory and Applications in Logic and Complexity, Dagstuhl-Seminar-Report; 5, 14.-18.1.1991 (9103)

B. Becker, Ch. Meinel (editors):
Entwerfen, Prüfen, Testen, Dagstuhl-Seminar-Report; 6, 18.-22.2.1991 (9108)

J. P. Finance, S. Jähnichen, J. Loeckx, M. Wirsing (editors):
Logical Theory for Program Construction, Dagstuhl-Seminar-Report; 7, 25.2.-1.3.1991 (9109)

E. W. Mayr, F. Meyer auf der Heide (editors):
Parallel and Distributed Algorithms, Dagstuhl-Seminar-Report; 8, 4.-8.3.1991 (9110)

M. Broy, P. Deussen, E.-R. Olderog, W.P. de Roever (editors):
Concurrent Systems: Semantics, Specification, and Synthesis, Dagstuhl-Seminar-Report; 9, 11.-15.3.1991 (9111)

K. Apt, K. Indermark, M. Rodriguez-Artalejo (editors):
Integration of Functional and Logic Programming, Dagstuhl-Seminar-Report; 10, 18.-22.3.1991 (9112)

E. Novak, J. Traub, H. Wozniakowski (editors):
Algorithms and Complexity for Continuous Problems, Dagstuhl-Seminar-Report; 11, 15-19.4.1991 (9116)

B. Nebel, C. Peltason, K. v. Luck (editors):
Terminological Logics, Dagstuhl-Seminar-Report; 12, 6.5.-18.5.1991 (9119)

R. Giegerich, S. Graham (editors):
Code Generation - Concepts, Tools, Techniques, Dagstuhl-Seminar-Report; 13, 20.-24.5.1991 (9121)

M. Karpinski, M. Luby, U. Vazirani (editors):
Randomized Algorithms, Dagstuhl-Seminar-Report; 14, 10.-14.6.1991 (9124)

J. Ch. Freytag, D. Maier, G. Vossen (editors):
Query Processing in Object-Oriented, Complex-Object and Nested Relation Databases, Dagstuhl-Seminar-Report; 15, 17.-21.6.1991 (9125)

M. Droste, Y. Gurevich (editors):
Semantics of Programming Languages and Model Theory, Dagstuhl-Seminar-Report; 16, 24.-28.6.1991 (9126)

G. Farin, H. Hagen, H. Noltemeier (editors):
Geometric Modelling, Dagstuhl-Seminar-Report; 17, 1.-5.7.1991 (9127)

A. Karshmer, J. Nehmer (editors):
Operating Systems of the 1990s, Dagstuhl-Seminar-Report; 18, 8.-12.7.1991 (9128)

H. Hagen, H. Müller, G.M. Nielson (editors):
Scientific Visualization, Dagstuhl-Seminar-Report 19, 26.8.-30.8.91 (9135)

T. Lengauer, R. Möhring, B. Preas (editors):
Theory and Practice of Physical Design of VLSI Systems, Dagstuhl-Seminar-Report 20, 2.9.-6.9.91 (9136)

F. Bancilhon, P. Lockemann, D. Tsichritzis (editors):
Directions of Future Database Research, Dagstuhl-Seminar-Report 21, 9.9.-13.9.91

H. Alt , B. Chazelle, E. Welz (editors):
Computational Geometry, Dagstuhl-Seminar-Report 22, 07.10.-11.10.91 (9137)

F.J. Brandenburg , J. Berstel, D. Wotschke (editors):
Trends and Applications in Formal Language Theory, Dagstuhl-Seminar-Report 23,14.10.-18.10.91 (9142)

H. Comon , H. Ganzinger, C. Kirchner, H. Kirchner, J.-L. Lassez , G. Smolka (editors):
Theorem Proving and Logic Programming with Constraints, Dagstuhl-Seminar-Report 24, 21.10.-25.10.91 (9143)

H. Noltemeier, T. Ottmann, D. Wood (editors):
Data Structures, Dagstuhl-Seminar-Report 25, 4.11.-8.11.91 (9145)

A. Borodin, A. Dress, M. Karpinski (editors):
Efficient Interpolation Algorithms, Dagstuhl-Seminar-Report 26, 2.-6.12.91 (9149)

B. Buchberger, J. Davenport, F. Schwarz (editors):
Algorithms of Computeralgebra, Dagstuhl-Seminar-Report 27, 16..-20.12.91 (9151)