The Bottom-Up Design of a Prolog Architecture

Richard Carlson

Computer Science Division
University of California, Berkeley, 94720

May 22, 1989

ABSTRACT

This report describes a set of tools developed to help define an efficient computer architecture for executing Prolog programs. These tools decompose Prolog source code into basic register-transfer level operations, compact these operations into more complex instructions while enforcing various architectural limits within the processor, and simulate and gather statistics on the resulting programs. This report also describes a group of Prolog benchmarks that have been analyzed by this set of tools, and presents some necessary features of an efficient Prolog architecture that were revealed by the analysis of these benchmarks.

This research was partially sponsored by Defense Advanced Research Projects Agency (DoD) and monitored by Office of Naval Research under Contract No. N00014-88-K-0579.

Contents

1. Introduction and Overview

Computer architects today face a dilemma. A processor can be designed to execute a wide variety of programs written in a wide variety of programming languages equally well, or it can be given specialized features, and be tailored to a specific application or set of applications. Of course, there are advantages and disadvantages to each approach. For a computer that will have to support a wide range of users and tasks, a specialized processor is probably inappropriate—its specialized functionality will benefit a small set of tasks. Furthermore, this benefit does not come for free: the increased processor complexity may slow the entire processor down, *reducing* the performance of tasks that cannot take advantage of the special features.

On the other hand, many computer systems will be called upon to perform only some very small subset of all the widely-varying jobs that computers can handle. In any particular computer system, if a set of very common operations can be greatly sped up by using specialized hardware, the system's overall performance will be improved, even if the less-common operations execute slightly slower than on a general-purpose processor.

1.1 Background

This project examines one particular "job subset" that a computer system might be expected to handle very efficiently: executing programs written in a *logic* programming language—specifically, the high-level language **Prolog** [CM81]. Many of Prolog's "typical" operations are very different from the operations performed in procedural languages such as C. For example, program execution does not flow directly and explicitly from procedure to procedure; rather, Prolog can automatically search through a group of alternative clauses—trying one clause, finding that it cannot be satisfied, backtracking to undo the effects of the partial execution of that clause, then moving on to try the next possible clause—until a clause is found that can be completely satisfied. And variables can be passed *unbound* from procedure to procedure, and then become instantiated by **unification**, a very general form of pattern matching. In the general case, unification can become very complicated—for example, if two complex structures are to be unified, each element or substructure must be recursively unified, binding previously-unbound variables to specific values or to other unbound variables.

A basic challenge is just how these high-level operations can actually be implemented in a processor. One of the first successful attempts at defining an efficient Prolog-specific processor was described by D. Warren [War83]. The "Warren Abstract Machine" specifies a set of primitive operations that Prolog programs can be decomposed into, and a basic processor architecture that is required to implement these operations. This model was so successful that most of today's general-purpose processor Prolog implementations are based on variations of the WAM instruction set. Prolog source code is translated into WAM instructions, and then the intermediate-level WAM code is compiled or interpreted in the machine language of a "general-purpose" processor. Several commercially successful Prolog products, such as Quintus Prolog, do exactly this. But by constructing the Prolog primitive operations out of general-purpose processor instructions, Prolog programs typically execute much slower than counterparts written in languages like C. Thus, in order to achieve the maximum possible performance from a Prolog system, it seems necessary to include some features in the processor that are designed specifically for the needs of Prolog.

With this goal, the Aquarius group at U.C. Berkeley designed and implemented a processor specifically tailored to execute Prolog. This Programmed Logic Machine (PLM) uses a slightly-modified version of WAM code as its machine language. Common operations in Prolog, such as unification, are implemented in microcode in the processor, rather than in a software subroutine for a general-purpose processor. And data paths in the PLM are designed to allow parallelism in data transfers during the execution of the processor's microcode. This project was successful in achieving an improvement of an order of magnitude over other Prolog systems at the time [Dob87].

Since that time, improvements in technology and compilation techniques have continued to improve Prolog performance. But to attain even higher levels of performance, it is useful to re-examine the WAM instruction set, identify its strengths and weaknesses, and attempt to improve upon it. The WAM instruction set makes a relatively high-level machine language. This has the advantage that compiled programs are very small, and correspondingly, that little time is spent in fetching machine-language instructions. Unfortunately, it also has the disadvantage that it is often inefficient, even when implemented in hardware. The complicated specialized instructions are "overkill" in some situations, wasting time in handling general cases. And, with these high-level machine instructions, it is often difficult to overlap different tasks and execute them in parallel, even if there are no data-dependency constraints that would make doing so incorrect.

1.2 Approach

To help overcome the limitations of the WAM instruction set, this project takes a different, "bottom-up" approach to the development of an instruction set for Prolog. The memory utilization, register usage, and models of program execution are still based upon the WAM, but instead of mapping Prolog source constructs into machine-language equivalents (the "top-down" design of the WAM), this project decomposes several typical Prolog programs into their very basic (register-transfer level) operations. Then, these basic operations are examined to see how they can best be grouped or composed into primitive machine-level instructions.

As the first step in the bottom-up design process, a group of "typical" programs must be selected that will be analyzed. Naturally, the larger the set of these "benchmark" programs, the more representative the analysis will be. After analyzing enough benchmarks, it should be possible to extrapolate the analysis to estimate the range of behavior of a larger set of "typical" Prolog programs.

But analyzing, say, hundreds of benchmarks is beyond the scope of this project. Instead, the project examines a handful of benchmarks that represent a variety of types of operations that are common in Prolog programs. These benchmarks were chosen to represent the various types of common operations found in programs. However, it is not necessary (or perhaps even possible) to characterize the *exact* procedures that "typical" programs contain. For example, the benchmarks include programs that manipulate both lists and structures, since these are two of the primitive data types provided by Prolog. Even if few programs perform the *exact* manipulations done by these benchmarks (such as concatenating two lists), many programs will perform manipulations of the same general *type* (such as scanning through a list of elements to find a particular one), and many of the necessary primitive operations and capabilities will be similar. Therefore, if these list-manipulating benchmarks can be efficiently executed, it is hoped that most other list-manipulating programs will also be relatively efficient.

1.3 Software Tools

Figure 1.1 shows the overall organization of this project, and the functions of the software tools written and used to complete it.

After a benchmark is selected, it must be decomposed into its primitive register-transfer level (RTL) operations. One option is to manually compile the benchmark into its low-level operations. This approach has one major advantage: a human is able to *understand* the benchmark programs, and can make decisions and simplifications that no automatic process (at least with current technology) could determine. The overwhelming disadvantage of this approach, however, is

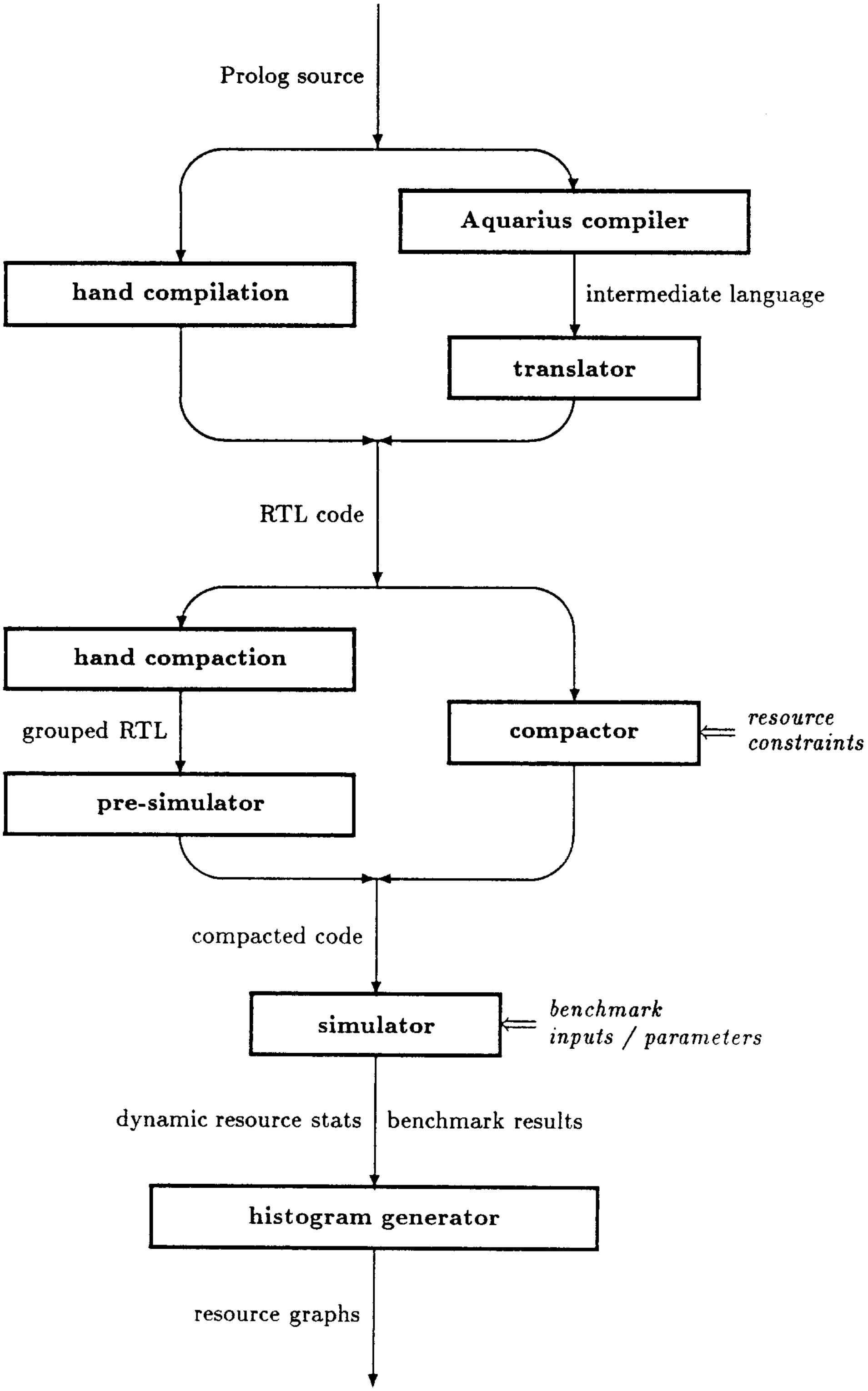


Figure 1.1: Organization of project tools.

that (for any non-trivial program) hand-compilation is extremely tedious, error-prone, and time-consuming.

The alternative method used in this project first uses a Prolog compiler being developed for the Aquarius research group[VRss] to compile the Prolog source into "intermediate-level" code. This code is at a lower level than WAM code, but it is not tailored to a specific microarchitecture, either. Its level of detail is at an intermediate point, that can be efficiently translated into various machine languages, as long as hardware support is available for handling tagged data, etc. After the compilation, a relatively straightforward translator program can decompose the intermediate-level code into its primitive RTL operations, as used in this project.

Once the RTL representation of the benchmark is obtained, the RTL operations must be grouped into "instructions," or "compacted." Again, there are two alternative methods to accomplish this compaction, a manual method and an automatic method. Manually, a programmer must examine each RTL operation, and determine how it can best be grouped with other RTL operations without damaging the correct operation of the program and without violating any resource limitations or constraints. Again, this process is time-consuming and error-prone. But a much more serious drawback is that the entire process must be repeated if the resource constraints are to be changed. For example, if a benchmark is hand-compacted using only single-word bandwidth to memory, and if the programmer wants to compare the performance of a double-word path to memory, the entire program must be hand-compacted again. Clearly, comparative analysis of widely-varying resource limitations is practically impossible with this technique. However, this method does offer a valid reference point of how tightly a benchmark's RTL operations can be compacted using human ingenuity, to which the automatic compaction method can be compared.

The automatic method of compacting RTL operations into instructions uses a microcode compactor written in Prolog. Aside from being faster and easier than hand-compaction, this compactor allows different resource limitations to be easily specified before compacting a benchmark. Thus, a wide variety of basic architectural features, and their effects on performance, can be examined relatively quickly.

Because the compactor must keep track of which resources are used in each instruction anyway, it includes this *static* resource usage information in its output. The instruction simulator then produces its *dynamic* resource usage statistics by using the static information provided by the compactor, rather than by recalculating the information as each instruction is executed. Note, however, that since the simulator always assumes its input contains resource usage information, this information must be provided even for hand-compacted programs. A "pre-simulator" program is used to add this information to programs that are not automatically compacted, and calculates resource usage in exactly the same manner as the compactor program.

After the compacted code is obtained, a simulator is used to simulate its execution, both to verify the correctness of the benchmark and to gather dynamic resource usage statistics. To transform the results into an easily-readable form, a histogram generator produces a graphical representation of resource usage from the raw statistics output by the simulator. For maximum flexibility, the simulator allows different parameters, or input arguments, to be given to the benchmark at run time. For example, the **concat** benchmark is examined in four different modes of operation—rather than going through the entire process of compacting four separate benchmarks, the simulator allows the same compacted code to be used for each test, simply supplying the different tests with different initial arguments.

Among these software tools, the Aquarius compiler is the only one not specifically written for this project.

1.4 Report Organization

The following section of this report, Section 2, gives a broad overview of the Prolog language, including some of its unique features and how they affect a Prolog computer's architecture.

Sections 3 and 4 describe the tools written for this project in more detail: the microcode compactor, being the most complex of the tools, is discussed by itself in Section 3; the other tools are described in Section 4. Section 5 identifies the Prolog benchmark programs that this project examines, and indicates why these benchmarks were chosen. The results of the analysis of these benchmarks are presented in Section 6. And, for the interested reader, the source code to the programs written for this project is given in the appendices.

2. The Prolog Execution Model

This report will not attempt to explain all of the details of Prolog execution or the WAM model upon which this project was based; these topics are outside the scope of this project report. However, this section will briefly summarize some of the most important aspects of the WAM model as they pertain to this project, and indicate some of the simplifying assumptions that have been made. A reader who wishes to learn more details is invited to refer to the bibliography at the end of this report.

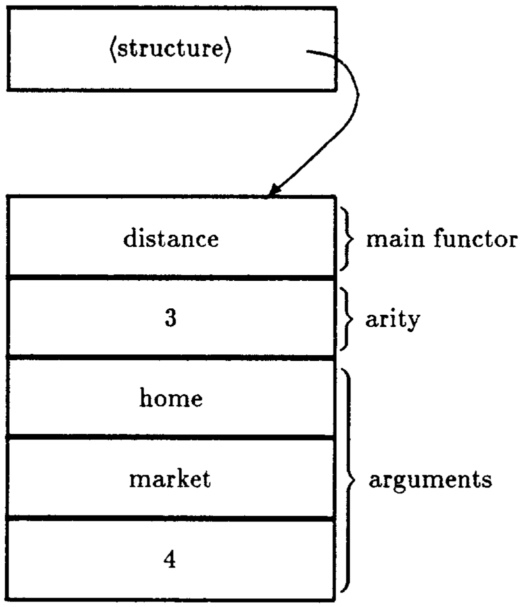
2.1 Data Representation

The WAM model includes four basic data types: constants, structures, lists, and variables. These types are similar to those found in other high-level programming languages. Any complex data structure must (and can) be built from a combination of these basic types.

- **Constants** represent individual, indivisible data objects. Examples of constants include `42`, `65.28`, and `fred`. Note that constants can be either integer or floating-point numbers, or **atoms** (unique, indivisible data objects). Constants that have numerical values may participate in the usual arithmetic and (for integral values) logical operations. In this project, it is assumed that all constants occupy exactly one (32-bit) word in memory. (Note that an atom such as `fred` will usually be implemented as a pointer to a block of memory that contain additional information about the constant, such as its printable name "`fred`". These details are not important to this project, since the usual operations on constants need to use only the one-word constant value. The only requirement is that this value be unique for each different constant.)

- **Structures** are used to group pieces of data, so that they can be treated as a unit. Each structure has a **functor**, which is the name assigned to the structure as a whole, and a certain number of **arguments** that represent the collected data in the structure. (The **arity** of a structure equals the number of arguments it has.) An example of a structure is `distance(home, market, 4)`; this structure has the functor `distance` and three arguments: `home`, `market`, and `4`. In this example, all three arguments are constants, but note that arguments are free to be any other type as well, including other structures. Thus, arbitrarily complex data structures may be created using this "structure" data type.

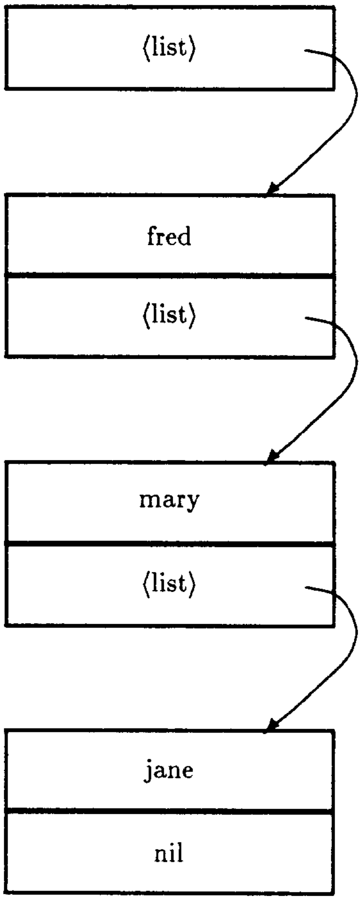
There are several valid methods for representing a structure in memory; the following diagram indicates how the method chosen in this project would represent the above example structure:



Thus, any time the structure as a whole is used, it can be referred to by a single-word pointer. To manipulate the individual arguments, that pointer must be followed to the "body" of the structure. This body contains the structure's functor, arity, and arguments in sequential memory locations. (Because this representation includes the arity in the structure body, no special "end-of-structure" marker is needed to indicate the end of the argument list, as is needed in some other implementations.)

- **Lists** are just a special form of structures that implement a linked list data type. Lists are treated as a separate data type (and are given a special notation in Prolog source code) because they provide a simple but useful structure in many different applications. One "list element" of a list is like a structure of arity 2, whose first argument is an arbitrary data element, and whose second argument is a pointer to the next sequential list element. However, to save space and time, the functor and arity that would be required in a normal structure element are not explicitly stored in memory for a list element. To signify the end of the list, the second argument is set to a special constant, `nil`.

An example list can be written as `[fred, mary, jane]` in Prolog source code. This list of three elements would be represented in memory as:



Different list elements can be located anywhere in memory, not necessarily contiguously or even in ascending order. (Some other implementations allow "cdr-coded" lists, in which list elements located contiguously in memory can omit the explicit pointers to the next element. Such lists are not used in this project.)

• **Variables** represent specific and unique (but possibly unknown) objects in Prolog, following predicate logic. In other words, unlike many other high-level languages in which variables can be updated many times, a Prolog variable represents a single value throughout a program (although varying amounts of computation will have to be done before the program knows just what that value is). This makes Prolog a single-assignment language.

At the Prolog source level, unbound variables can be used to obtain indirection, without the explicit programmer-specified pointer operations necessary in other high-level languages. For example, if two unbound variables are bound to each other, binding either of those variables will automatically cause the other to receive that same value. (This feature is used extensively in the microcode compactor program.) At the machine level, an object of type "variable" can be used either to obtain indirection (a "variable" pointing to another data object has the value of that object), or to indicate that a value is not yet known (by convention, a "variable" pointing to itself is "unbound").

2.2 Memory Sections

The benchmarks in this project divide system memory into four primary regions: a heap, a trail, an execution stack, and a unification stack.

- The **heap** is the section of memory used to hold permanent or semi-permanent information. Any data that cannot be held in a register (including variables, and the bodies of structures and lists) must be written into the heap.

- The **trail** is made necessary by Prolog's ability to **backtrack**. Recall that a programmer can specify several alternative solutions for any predicate. If the first alternative cannot be satisfied, Prolog will automatically back up to the last point at which an alternative solution may exist, and try one of the alternatives. The point that Prolog returns to and starts trying a new alternative from is called a "choice point." Note, however, that backtracking is more complicated than simply causing program execution to continue at an alternate location. The first alternative may have changed many data structures before it failed; these changes will not be applicable to the other alternatives, because each alternative must see exactly the same program state when it begins to execute. Therefore, before a new alternative predicate can start to execute, all data structures must be restored as they were before the first alternative predicate began to execute.

This restoration process is eased somewhat by Prolog's single-assignment rule: the only way data can be "changed" is by giving a value to an unbound variable. Therefore, any restoration simply involves "unbinding" a set of variables. The trail is a stack that lists the address of every variable as it gets bound. By marking the size of the trail stack every time a choice point is created, whenever backtracking must occur it is simple to determine which variables were bound since the last choice point. This is exactly the set of variables that must be unbound before the next alternative can begin to execute. After unbinding these variables, they can be "popped off" of the trail stack, so that the size of the trail does not grow without bound.

- The **execution stack** holds information about the current state of program execution. It is similar in function to the stack used in other programming languages, that holds information about subroutine calls and how to return from them. But in Prolog, the stack must also hold information about how to backtrack if a predicate fails. Therefore, the Prolog stack must contain two types of information: "stack frames" that are used to return to the calling predicate when (and if) the current predicate succeeds; and "choice points" that indicate the next alternative to be tried if the current predicate fails.

Each stack frame contains the address to return to if the current procedure succeeds, and a pointer to the previous stack frame (to be restored when returning to the calling procedure). A stack frame can also reserve space for a procedure's local variables.

Each choice point contains the values of all registers that must be restored if failure and backtracking occur. These include: argument registers; the current values of the heap, trail, environment, and frame pointers; the current return (success) address; the address of the next alternative procedure to try if this one fails; and a pointer to the previous choice point (so it can be activated when the current one becomes inactive). Note that the failure of a clause renders useless any structures placed on the heap since the last choice point; restoring the heap pointer during failure processing automatically "cleans up" the heap by deallocating such structures. Also note that, before restoring the trail pointer, the failure subroutine must specifically unbind any variables listed in the trail stack.

- The **unification stack** is used only during the process of unifying complex data types (structures or lists). As previously mentioned, structure arguments can be other structures, and any or all of the arguments can be unbound. Thus, unifying two structures can become arbitrarily complex as arguments that are other structures must be recursively unified. The unification stack maintains the status information needed to perform (and complete) such recursive unification steps.

Note that a few other memory sections are necessary to support all of the features of Prolog, such as dynamically adding or removing program code. No benchmarks using these features were simulated in this project, so this project's model was simplified by omitting those other memory sections. In many Prolog programs, these more exotic features of Prolog are relatively rarely used, so the analysis done in this project should be fairly representative. However, this project's models might not accurately reflect the needs of those applications that rely heavily on these features.

3. Microcode Compactor

As mentioned above, one of the largest parts of this project is the microcode compactor. It accepts as input a list of primitive register-transfer level operations. It then combines, or compacts, them into "instructions" (groups of operations that can be carried out concurrently). It finally emits this list of instructions as its output. This section explains the microcode compactor in more detail.

3.1 Combining Operations in a Basic Block

The primary goal of the compactor is to squeeze as many operations into each instruction as possible, thereby reducing the execution time of the program. In other words, this program compacts its input microcode in terms of *execution time*, rather than storage space.

Of course, in the process of compacting a program, the program's behavior cannot be changed. To ensure this, the compactor must maintain and use data dependency information, which keeps track of all locations (memory and registers) that are read from or written to by each operation. Specifically, if the original operation list includes operations **A** and **B**, and **B** occurs some time after **A**, then we have the following possibilities:

- If **A** reads from a location and **B** writes to the same location, then we have a *read-write* dependency, and **B** can be placed into an instruction *no earlier* than the instruction that contains **A**. If this constraint were violated, **A** would incorrectly read the *new* value from **B**. Note that this compactor allows **A** and **B** to be placed into the *same* instruction, modeling the behavior of registers on most modern processors (that is, a written register gets updated at the *end* of a clock cycle, so a read of that register during the same clock cycle will return the old value).

- If **A** writes to a location and **B** reads from the same location, then we have a *write-read* dependency, and **B** must be placed into an instruction *later* than the instruction that contains **A**. If this constraint were violated, **B** would incorrectly read the *old* value instead of the updated value from **A**.

- If both **A** and **B** write to the same location, then we have a *write-write* dependency, and **B** must again be placed into a *later* instruction than **A**. If this constraint were violated, later operations would read the value written by **A** even though **B**'s value should have superceded it.

Note that if **A** and **B** only read from the same location, or if **A** and **B** access no common locations at all, then no constraints are placed on the relative scheduling of **A** and **B**, and it is perfectly legal for **B** to precede **A**.

In general, each operation will access several different locations, and so each operation will likely have several dependencies (of different types) on various other operations. The set of all dependencies within a list of operations can be viewed as an *acyclic* directed graph. Figure 3.1 indicates a possible dependency graph. In this example, the expression "`a <-- b + c`" denotes adding the contents of `b` to the contents of `c` and storing the result in `a`. Thin lines point to an operation that must be scheduled *no earlier* than another operation, and thick lines point to an operation that must be scheduled *later* than another operation.

A slight complication arises when indexed memory locations must be included in the dependency analysis, because the memory address—the sum of the contents of a base register and an integer offset—is not known at compaction time. One solution (and the solution used by this

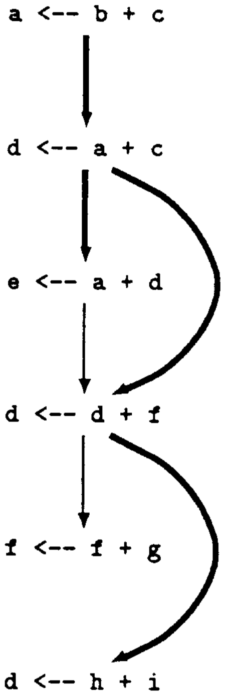


Figure 3.1: A typical dependency graph.

compactor) is to group memory locations according to which base register is used. For example, all memory addresses accessed by any offset from the e base-register will be treated as one "location" by the data dependency code. In Prolog, all base registers are used to refer to disjoint sections of memory, so this solution will provide a somewhat conservative, but correct, dependency analysis. There is actually one situation in which this analysis may not detect a dependency. If a variable is placed into one memory section that points into a different memory section. In practice, this happens and causes compaction problems only very rarely, and so the compactor does not attempt to detect this special case (if problems do arise, manual constraints can be added to the operation list).

The rules for scheduling an operation are conceptually very simple. An operation **A** is ready to be scheduled into an instruction only if

(1) all operations that **A** must be scheduled "later than" have been placed in some previous instruction, and

(2) all operations that **A** must be scheduled "no earlier than" are placed either in the current instruction or in some previous instruction.

The compactor operates by packing as many operations as possible into one instruction, and then proceeding to the next sequential instruction. It implements the above rules by maintaining, for each operation, lists of "dependent operations," other operations that have some sort of dependency on it. Then, when scheduling any operation, all of its dependent operations are checked; if scheduling the operation has satisfied the last remaining dependency of a dependent operation, that dependent operation becomes "ready to schedule." Note that, depending on the types of its dependencies, an operation that has become ready to schedule might be able to be placed in the *current* instruction, or it might have to wait until the following instruction.

In addition to the dependencies that the compactor automatically determines, the programmer might wish to specify additional constraints. Therefore, a special syntax in the input microcode can specify extra constraints of the "later" or the "no earlier" varieties. Additionally, for the generality to handle some special situations, there is also a "forced" type of dependency, where one operation is forced to be scheduled into an instruction *exactly* some fixed number of instructions after another operation.

The above describes an "unlimited" type of compaction, in which the only limitation on how tightly the code can be compacted is that no data (or manually-specified) dependencies are violated. When operated in this mode, the compactor produces the most highly-compacted code, which, subject to the limitations of the compactor, represents the maximum parallelism available in the input program.

However, the compactor can also be programmed to take some practical limitations of a real processor into account. For example, a maximum number of simultaneous ALU operations can be given, and the compactor will never schedule more than the given number of ALU operations into any individual instruction, even if no data dependencies would be violated. This ability to limit resources provides a key instrument in determining the "optimum" number of the various resources that the processor should include. This optimum is the minimum number of each type of resource that does not significantly increase a benchmark program's execution time. Each resource can be limited more and more until a "knee" in the program's execution time graph is reached, and further limitations in the resource cause much more dramatic increases in execution time. An auxiliary file provides the basic compactor program the information it needs to count and limit resources, so examining a different set of resource limits is as simple as running the compactor with a different auxiliary file.

3.2 Trace Scheduling

So far, we have been considering only operations that manipulate data, such as adding two numbers and saving the result. Another major type of operation changes the flow of execution of a program is conditional or unconditional branches. And, unfortunately, these branches considerably complicate the compaction process.

The simplest way to handle branch operations is to divide the input operation list into subsequences that contain no branch operations, except for one branch operation at the end of the subsequence. These subsequences are often called "basic blocks," and each basic block can be compacted independently of all other blocks in the method already described. But although this method is relatively simple, its inability to combine operations from different blocks is a severe disadvantage. Considering that branch operations are relatively frequent, and that there are therefore very few instructions within most basic blocks, it is likely that only very limited parallelism would be obtained by this method.

To obtain more opportunities for combining operations, this compactor uses a more advanced compaction method, which is based on trace scheduling. This approach chooses one possible path of execution through the input program, from beginning to end, and compacts the entire path, or trace, as a unit. As long as no data dependencies are violated, operations from different blocks can be freely combined. After one trace has been compacted, another is chosen and compacted, until compacted instructions for all possible paths of execution have been created.

The complications of this method arise while compacting those traces in which some of the operations have already been compacted into instructions along some other trace. In most cases, it would be possible to completely compact each trace independently of all others, but this approach would duplicate much compaction work, and would greatly increase the size of the compacted program, since each operation would possibly appear along many different traces. Furthermore, special consideration must still be given to loops, since (in general) the number of iterations is not known at compaction time, and the compacted code must allow for an arbitrary number of iterations.

For these reasons, this compactor tries to re-use as much previously-compacted code as possible when compacting a new trace. But because the operations in the previous traces have probably been moved across the original basic block boundaries, an operation that we want to schedule in our new trace—even if it appears in another trace—may occur in a position that is unusable. That is, the new trace cannot jump into the old trace to execute a needed operation, because doing so would also cause operations that do *not* occur along the current trace to be executed. Thus, the compactor must maintain extra data structures that record which blocks are represented in each of the compacted instructions.

3.3 Compactor Operation

The specific actions of the compactor are illustrated in Figure 3.2.

The compactor performs some pre-processing before beginning the trace scheduling itself. First, the innermost loops in the input program are "unrolled" once. That is, any loop the body of which contains no other branches, has the code in its body duplicated. This technique can improve the compaction of an innermost loop, by allowing some operations from the $(n-1)$th iteration of the loop to be overlapped with other operations from the nth iteration, as long as no data dependencies are violated. If loop unrolling is not done, the loop boundaries can artificially limit concurrency. Second, the source representation of the RTL program is expanded into the data structures used internally in the compactor. During this process, the basic blocks (i.e., sequences of loop-free code) in the program are also determined, and labels and branches are modified to refer to blocks rather than individual operations. Since the number of operations far exceeds the number of blocks in a typical program, this process simplifies and speeds up the trace scheduling.

In the trace scheduler itself, the first operation that is performed is selecting one possible execution path as the next trace to be compacted. Each execution path begins at the start of the program or at the start of an uncompacted block, and continues until the path encounters a block that has already been compacted or the end of the program. When a conditional branch is encountered, the compactor must choose which path will continue the current trace. Later trace scheduling passes will compact whichever paths are not chosen. However, the later traces are likely to be shorter—containing fewer operations, those later traces are likely to have more limited parallelism and, therefore, might be compacted less tightly. Thus, it would be beneficial if the compactor could choose the most common direction of a conditional branch to first be compacted. In practice, determining this "most common" branch direction would be very difficult, so the compactor uses a simple heuristic to guess which it is: in a list of conditional branch directions, the branch direction listed first is compacted first. All of the hand-coded benchmarks use this fact, and list the most common branch direction first. The compiled benchmarks, however, do not optimize the order of conditional branches; perhaps improved compilation techniques will eventually allow such optimizations.

While choosing a trace to compact, the compactor maintains lists of which blocks, and how many operations from each block, are represented in the trace. When another trace wants to branch to one of the blocks that were originally compacted in the current trace, this block information is needed to determine just where the branch should point. For example, consider that the current trace contains operations from four blocks, 1, 2, 3, and 4. Identifying each operation by its block number, suppose that the list of operations

1		2*
1		1
2		3*
2	is reordered as	1
2		2 ⇐
3		2
3		3

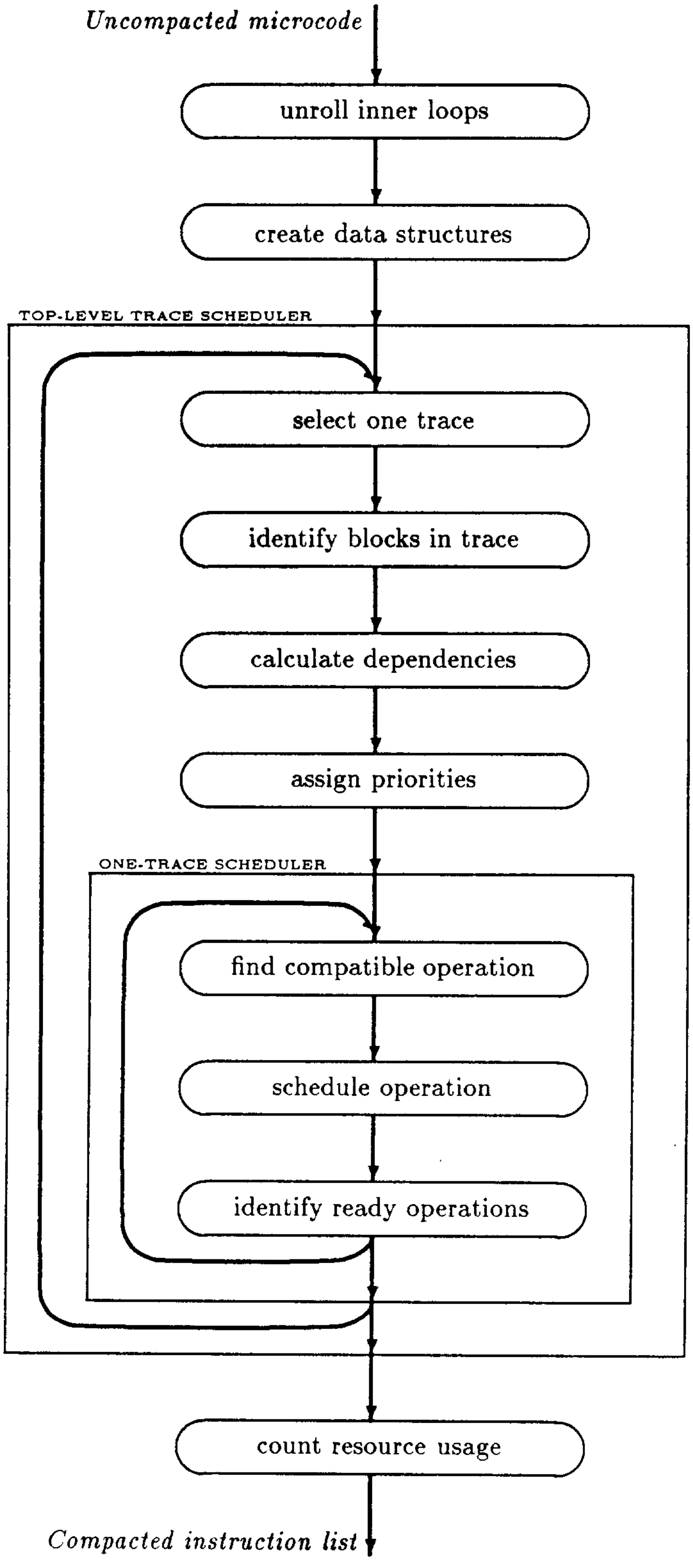


Figure 3.2: Operation of the microcode compactor.

Then, if another trace wants to branch to block 2, it *cannot* branch to the first operation from block 2 in the reordered program—doing so would also execute the operations from block 1, which is not legal. Instead, the new trace must include duplicate copies of the operations marked with an asterisk, and then jump to the "safe" point marked by the arrow. The block list structures allow the compactor to identify operations (such as those marked with asterisks above) that may need to be duplicated, and to identify the safe entry point for each block.

The next step in the trace scheduling process is to calculate the data dependencies among the instructions in the current trace. In addition to the read and write dependencies previously discussed, this section of the compactor must also

- add dependencies to prevent reordering of the input and output pseudo-operations (for example, we wouldn't want the user to receive a prompt for input before the text explaining what is wanted is printed out);
- add dependencies to prevent *any* operation from moving before a branch operation—since live register analysis is not done, any such movement is potentially illegal; and
- preserve any user-specified dependencies from the source program.

Then, after all dependencies have been determined, a scheduling priority is assigned to each operation. The current algorithm for determining priorities is relatively simple: all of an operation's dependents are examined to find the one with the highest priority value, then the current operation is assigned a priority value one greater than that maximum value. In other words, an operation's priority is proportional to the number of operations waiting for it to be executed. The priority is important when there are resource limitations, because out of several operations that can *logically* be scheduled into an instruction, only a few of those operations can *actually* be scheduled before exhausting the available resources. In such cases, the highest-priority operations are scheduled into an instruction first.

With all of the information calculated for each of the operations along a trace, these operations are scheduled by the single-trace scheduler. This section of code attempts to pack as many operations as possible into each instruction, and then move on to the next instruction. At any point, the single-trace scheduler has a list of logically schedulable operations. From highest to lowest priority, each operation is checked to see if it is "compatible" with the current instruction; that is, if the resources it needs are still available. If so, the operation is scheduled; if not, the next operation is checked. When no more operations will fit into the current instruction, the scheduler moves on to the next instruction slot. Additionally, any time a new operation is scheduled, any or all of its dependents might become ready-to-schedule, and so each dependent operation must be checked. If any has no more unsatisfied dependencies, it is now ready to schedule.

When the single-trace scheduler has finished scheduling all of the operations along its trace, the top-level trace scheduler selects another uncompacted trace to work on. After all possible execution paths in the program have been compacted, the top-level trace scheduler exits. At that point, the number of resources used in each instruction is counted, and the compacted instruction list is written out (in a Prolog-readable form for the benefit of the instruction simulator).

There are many other techniques that a compactor can use to produce more efficient code. In fact, a compactor can become almost arbitrarily complex as it performs more analyses of larger sections of the input program. The compactor used in this project was made only complicated enough to produce reasonably compact programs, in which the limiting factors in program speed are the data dependency and resource constraints, not the sophistication of the compactor. The analysis section of this report shows that the compactor satisfies this requirement.

Nevertheless, perhaps the single largest improvement that could be made to this compactor would be the addition of live register analysis. This would give the compactor much more freedom to move operations past conditional jumps—in particular, if a register is assigned in one path of a conditional jump, but never used in any other paths, that register assignment could legally be moved before the conditional jump. See [DLSM81, Fis81, FLS81] for more details on advanced compaction techniques.

4. Other Software Tools

Of the programs written for this project, the microcode compactor is the most complex. However, each of the other programs is an important part of the total project. This section of the report describes these other programs in more detail.

4.1 Intermediate Code Translator

This translator converts the intermediate code produced by the Aquarius compiler into the operation list expected by the microcode compactor. Part of this translation involves a simple change of syntax. For example, the compiler would use the expression `tstr^h` to represent a structure at the address of register `h`. The translator would change that expression into `struct@h` to match the syntax expected by the microcode compactor.

At the instruction level, the compiler's intermediate language includes several primitive instructions such as `move(A,B)` to transfer the contents of `A` into `B`, and the instruction `add(R1,R2,R3)` to add the contents of registers `R1` and `R2` and store the result in `R3`. These simple instructions can be translated by a direct one-to-one conversion. For example, the above `move` instruction would be translated into `B<--A`, and the `add` instruction would become `R3<--R1+R2`.

However, several intermediate language instructions are slightly more complicated; they must be expanded into sequences of several RTL operations. For example, the compiler will generate an `allocate(N)` instruction when a stack frame, containing `N` local variables, is to be allocated. This instruction must be expanded into an RTL sequence such as `mem(s+N)<--e, mem(s+N+1)<--ret, e<--s, s<--s+N+2`, where the actions of saving the old environment pointer and return address, updating the environment pointer, and allocating stack space for this new frame are explicitly represented.

And, there are several intermediate language instructions that expand into so much RTL code that their RTL representations are implemented as "subroutines". These include the `try` instruction that creates a choice point (and saves the current processor state) on the stack, and the `unify(A,B)` instruction that attempts to unify its two arguments.

In general, even the more complicated intermediate language instructions are straightforward enough that the translator can generate the "optimal" equivalent RTL operations. However, there is one set of instructions for which the translator does not take full advantage of the compiler's information. These are the `try/retry/trust` instructions that allocate, modify, and remove choice points from the stack. When it generates these instructions, the compiler provides information telling exactly which argument registers must be included in the choice point. Unfortunately, the structure of the microinstruction simulator makes it difficult to efficiently save and restore only the necessary registers, so the translator always saves and restores *all* argument registers. There are also some instruction combinations that, together, generate non-optimal RTL sequences—the major offender in this respect is the combination of a `deref` instruction to dereference a variable, followed by a `switch` instruction to branch on the variable's type. Peephole optimization would often be able to combine conditional branch operations from the two operations. For the purposes of this project, however, these limitations were thought to be relatively minor.

4.2 Pre-Simulator

For those benchmarks that have been hand-translated directly into instructions, this program must be run before the benchmark can actually be simulated. Note the difference between hand-*compilation* and hand-*translation*: in the former, a programmer creates just a list of register-transfer level operations, which must still be compacted into instructions; in the latter, the programmer has also manually combined his register-transfer level operations into instructions.

In this latter case, since the human has already performed the code compaction, the program certainly does not need to be passed through the microcode compactor. However, the simulator assumes that its input contains, in addition to the register-transfer level operations, resource usage information. Without the pre-simulator, the programmer would have to manually add that usage information, a *very* tedious task. The sole responsibility of the pre-simulator is to add the resource usage information to its input operation list, using the same resource usage calculations as are done in the compactor. But, of course, the pre-simulator merely *counts* the resource usage; it cannot in any way limit that usage.

A second use of this pre-simulator is for debugging compiled programs. Because the microcode compactor duplicates, rearranges, and relabels the operations of its input list, it is usually difficult to follow the execution of a compacted program during simulation. By passing the compiled program through the pre-simulator instead of the compactor, each operation becomes an individual instruction (and so the resulting program takes many cycles to execute). The code is not changed and therefore can be easily debugged.

4.3 Simulator

This program accepts the compacted output from the microcode compactor (or the pre-simulator), and simulates its execution. This simulation, and verifying the correctness of the benchmark's output, are necessary steps in finding and eliminating bugs in the benchmark programs (and in the other software tools). If problems arise, the simulator can produce a step-by-step list of all instructions that were executed, so that the programmer can pinpoint the location(s) in the benchmark where things went astray.

However, a more important function of the simulator is the gathering of dynamic resource usage statistics—these statistics are the primary means for determining which features are beneficial in a Prolog architecture, and which features do not improve performance. As mentioned above, the compactor (or the pre-simulator) attaches static resource usage information to each compacted instruction. The simulator uses this information to create a *dynamic* list of resource usage, which properly gives more weight to those instructions that are executed most often—for example, the instructions in an inner loop.

The operation of the simulator involves two major phases. In the first phase, the program is read into the simulator's data structures. As a part of this process, a tree is created that maps every symbolic label name in the original program to its corresponding location in the simulator's internal instruction list. In this way, any branch can quickly locate its destination code without scanning through the entire program. For large programs, the initial overhead of this approach is infinitesimal compared to the time saved during the simulation.

The second phase of operation is the program simulation itself. A program is simulated one instruction at a time. The simulator is carefully written so that multiple operations within any one instruction will act as if they truly were executed in parallel (that is, any change in state made during an instruction is visible only after the end of that instruction). After each instruction is executed, its resource usage information is printed out, and, if the programmer wishes, a symbolic representation of the operations performed during that instruction can be printed. The programmer is allowed to select any combination of instructions (identified by execution cycle) to be symbolically printed, so, for example, he can ask that only instructions 1000 and higher be shown, if he is confident that earlier instructions are behaving correctly. Another important feature of this simulator is that it allows arguments to a program to be given when the benchmark is actually run. This allowed, for example, several different uses of the **concat** benchmark to be examined, using the same compacted code as input.

An auxiliary Prolog program is used in conjunction with the general simulator to provide model-specific information. This information includes how to represent and how to display the different data types for a particular execution model. Currently, there is one such auxiliary program that matches the data type representations described earlier in this report.

4.4 Histogram Generator

Finally, once we have a dynamic resource usage trace from the simulator, we must distill that information into an easily-understandable form. The histogram generator is a simple program that counts the number of times each resource was used a particular number of times, and produces a histogram of this information for each of the tracked resources. As a convenience, it also counts the total number of execution cycles and the total number of times each resource was used during execution.

Incidentally, being written in C, this is the only program written for this project that was not written in Prolog. The primary reason for this is that the histogram generator is mainly a numerical program. Unlike the other programs, there is very little symbolic information (that Prolog handles so well) to be manipulated. But there is, potentially, a *great deal* of numerical information to process, and, at least using current Prolog compilers, Prolog simply cannot compete with C in the execution speed of numerical programs.

5. Benchmarks

This section of the report briefly describes the benchmark programs that are analyzed in this project. All of these benchmarks are programs in (or variants of programs in) the Aquarius benchmark suite.[Hay89] The source code that was used for each of these benchmarks is included in the appendices. For those benchmarks that were hand compiled, the hand-compiled version of the benchmark is also included in the appendices.

5.1 `concat`

The first benchmark examined in this project is a standard list-manipulation program. `concat` accepts three arguments; by the definition of concat, the third argument is equal to the concatenation of the first two arguments, where all three arguments must be lists. Note, however, that this benchmark is *not* restricted to answering the question "what is the concatenation of lists **A** and **B**?" It can also answer the question "what list, when concatenated to **A**, results in the list **C**?," or even "what are all possible pairs of lists that, when concatenated, result in the list **C**?"

For this project, four of the possible uses of `concat` are examined. The following table lists the ways the benchmark was executed. An "input" argument is one that is bound to a list value when the benchmark is called; an "output" argument is one that is unbound when the benchmark is called, and that becomes bound to a list during the benchmark's execution.

benchmark version	argument 1	2	3
A	input	input	output
B	input	output	input
C	output	input	input
D	output	output	input

Tests **A**, **B**, and **C** all produce exactly one result, but they exercise different aspects of the `concat` benchmark (for example, if the first argument is an "input", only one of the two clauses for `concat` can apply, and it is possible to directly select the appropriate one; if the first argument is an "output", either clause can potentially succeed, depending upon how that first argument is bound). Test **D** is an explicit test of backtracking in this benchmark; 33 different results are generated, one for every possible way of obtaining a 32-element list by concatenating two other lists.

In all cases, the third argument is (initially, or after being calculated by the benchmark) a list of 32 elements. In all cases except **D**, the first two arguments are lists of 30 and 2 elements, respectively (for test **D**, the sum of the number of elements in the first two arguments is 32, for each solution).

5.2 `naive reverse`

The second benchmark is a different type of list manipulation program, that is also very common. This **`naive reverse`** benchmark reverses the elements of a list using an inefficient "naive" algorithm. Although there are more efficient ways to reverse a list, this method survives as perhaps the single most prevalent standard benchmark in the Prolog community. In conformance to the common standard, this project's **`naive reverse`** benchmark reverses a list of 30 elements.

Unlike `concat`, the implementation of this benchmark was written to work in only one mode. That is, the first argument is instantiated to a list, and the second argument is initially a variable, to be bound by **naive reverse** to the result of the reversal. This restriction greatly simplifies the code. In particular, there will never be a need to backtrack, so no backtracking capability is included in this benchmark.

Together, these first two benchmarks represent programs or sections of programs that make heavy use of lists—either creating or examining them. As mentioned in Section 2 of this report, list operations are very common in many programs, so this is an important class of operations to examine.

5.3 diff

The **diff** benchmark performs the symbolic differentiation of four different mathematical expressions. The operations that can be differentiated include addition, subtraction, multiplication, and division, raising an expression to an integral power, and taking the natural log of or exponentiating an expression. The benchmark makes no attempt to simplify the resulting expressions.

Like the **naive reverse** benchmark, this benchmark does not require any backtracking, so no backtracking code is included or used. The total execution time reported for this benchmark is the sum of the times taken to differentiate four different mathematical expressions. Each of these expressions uses different sets of differentiation rules, so the simulation results include operations from several of the differentiation clauses.

Whereas **concat** and **naive reverse** heavily manipulate lists, this **diff** benchmark makes heavy use of the other compound data type in Prolog, the *structure*. The benchmark both examines structures (to determine how to differentiate them) and creates structures (as the result of the differentiation). Again, this benchmark is an important one to examine, as it represents the (large number of) real programs that make heavy use of structures. The main limitation of this benchmark is that it uses only relatively small structures (one or two arguments). Larger structures might, for example, exhibit different patterns of memory behavior.

5.4 four queens

This benchmark is an adaptation of the popular 8-queens benchmark. That benchmark determines all possible arrangements of eight queens on a chessboard (an 8 × 8 grid of squares), such that no queen can attack any other queen. In other words, no queen is on the same row, column, or diagonal as any other. The only changes that were made to the original benchmark are that, in **four queens**, the chessboard is reduced from an 8 × 8 grid to a 4 × 4 grid, and we are only placing 4 queens on this board.

The benchmark was reduced from the 8-queens to the 4-queens problem to reduce the total simulation time. Note that this simplification should not significantly alter the behavior of the benchmark (it does not cause any part of the benchmark to be "skipped"); it simply reduces the repetition in the simulation results.

This benchmark backtracks until there are no more solutions. Thus, all possible sets of positions for the 4 queens are found (in this case of a 4 × 4 grid, there are only two valid sets of positions). This benchmark makes use of backtracking, simple structure operations, and simple mathematical expressions and comparisons. This benchmark was compiled using the Aquarius compiler.

5.5 mu math

This **mu math** benchmark is a slightly-modified version of a simple theorem-proving program based on the axioms of Hofstadter's mu math system; the original code is from Despain. The axioms of this system allow various sequences of the symbols **m**, **u**, and **i** to be generated from specific rules.

The original code proves the "theorem" **muiiu** in 5 steps (i.e., 5 applications of rules). To reduce the total simulation time without changing its basic behavior, this version of the benchmark proves a slightly different "theorem," **miuiu**, which can be proved in only 2 steps. This version of the benchmark also keeps track of which rules are used in the process of proving the theorem, to verify its correct operation. It makes heavy use of backtracking and list operations. This benchmark was also compiled using the Aquarius compiler.

6. Analysis

The final part of this project uses the tools described in Sections 3 and 4 to analyze the behavior of the benchmarks described in Section 5. This section of the report presents some of the results of this analysis.

6.1 Instruction Simulation

As previously mentioned, an "instruction" consists of one or more primitive operations that can be executed concurrently. In the present simulator, each instruction is assumed to require exactly one execution cycle. Thus, even an operation that (for example) loads a value from an indexed memory location, will be simulated in just one cycle, and the loaded value will be available to use during the following cycle. Most real architectures will impose more restrictions on operations like these. For example, the indexed memory address might have to be calculated one cycle before the memory is actually read, or the memory read itself might take more than one cycle, or memory accesses might not be allowed in adjacent cycles. Because this project is not tailored for any specific memory or processor implementation, such additional restrictions are not imposed. However, if more specific analysis were desired, additional restrictions like these could be easily specified during the compaction process.

This project's simulator and compactor also currently contain no notion of a "program counter", or of fetching instructions from memory. In other words, reading an instruction from memory will never conflict with reading or writing data to memory. In some processors, instruction and data memory references *can* conflict with each other. To model such processors accurately, these conflicts would also have to be modeled by the compactor. However, it is also possible to design processors that have separate paths to instruction memory and data memory. The current compactor behavior accurately models such systems.

6.2 Tracked Resources

The compactor and simulator are capable of tracking an almost unlimited number of different types of resources. To track and possibly limit a new type of resource, one must simply define how and when it is used by the various types of RTL operations, and implement these definitions in a few Prolog predicates that are used by the compactor.

For this project, however, the analysis was confined to three basic types of resources: ALUs, write ports to the processor registers (called "buses" in this section), and memory. The rest of this section defines these resources as they are interpreted in this project.

- An **ALU** is a unit that can accept one or two input operands, perform an arithmetic or logical operation, and produce a result in a single "instruction cycle." The possible operations are the usual single-cycle operations: addition, subtraction, and negation; bitwise ANDing, ORing, complementing; and left and right shifting. More complex operations such as multiplication or division must be coded as sequences of the more primitive operations.

One ALU operation is also charged for each equality or inequality test of two values. Prolog also makes frequent tests of the *type* of a value (to distinguish between constants, lists, structures, and variables). Because such type testing is so common, the compactor assumes that a single ALU test can distinguish between each of the different types.

Other relatively common operations are splitting an object into its separate type and value fields, and concatenating specified type and value fields to create a new object. Such operations are trivial to implement in hardware, involving only direct wire connections, so an ALU operation is *not* charged for these operations.

For the hand-compiled benchmarks, one ALU operation is charged for calculating a memory address that includes a displacement. However, analysis of the hand-compiled code showed that many instructions contained address offset calculations together with general ALU operations. For this reason, among others, the automatically-compiled benchmarks are analyzed including a separate address generator that is not counted in the number of general-purpose ALUs.

- A **bus** is a data path, internal to the processor, through which information can be written into a register or out to main memory. In a processor architecture that contains a register file, the number of "buses" would be equal to the number of write ports into the register file (but some of these buses would also have to be able to carry data out to main memory). Note that this definition does *not* include data paths that read information from a register and carry it to an ALU input. No limitations on reading from the registers are imposed.

Since each ALU is assumed to write directly to a register—rather than writing to its own intermediate accumulator—there must be at least one bus for each ALU, or else some ALU would not be able to store its result. But any direct transfer from one register to another also requires the use of a bus, so having *more* buses than ALUs allows direct transfers to take place even when all of the ALUs are in use.

- The **memory** system comprises any interactions between the processor and the outside world. During any one instruction, it is assumed that memory can be either read from or written to, but *not both*, regardless of the selected resource limitations. Furthermore, it is enforced that all memory references during one instruction refer to *contiguous* memory locations. Therefore, if the resource limitations specify that double-word memory accesses are allowed, memory locations `s` and `s+1` on the stack can be simultaneously accessed, but locations `s` on the stack and `tr` in the trail area cannot. Especially note that this restriction prevents locations `s` and `tr` from being simultaneously referenced even if *infinite* memory bandwidth is allowed, because the addresses are still not contiguous.

6.3 Compaction Efficiency

This first analysis examines the effectiveness of the microcode compactor. This analysis uses the hand-compacted versions of the `concat`, `naive reverse`, and `diff` benchmarks (in other words, the register-transfer level operations were manually grouped into instructions as efficiently as could be seen). This hand-compaction ensured that only a double-word memory read or write occurred in any one instruction; no other resources were limited. Then, the compactor automatically compacted these same benchmarks from the same register-transfer level code, using the same resource restrictions (that is, no more than a double-word memory read or write in an instruction, but unlimited ALU operations and bus transactions).

The following table compares the performance of the hand-compacted benchmarks with the automatically-compacted versions:

Benchmark	Execution time manual	automatic
`concat` A	96	92
`concat` B	221	217
`concat` C	1372	1339
`concat` D	1431	1334
`naive reverse`	1449	1478
`diff`	902	902

In all cases except `naive reverse`, the compactor-generated code executed slightly faster than the hand-compacted code. Even in the case of **naive reverse**, the compactor-generated code executed only slightly slower than the hand-compacted code. Because memory usage was the restricted resource, the hand-compacted code was carefully rearranged to perform as many memory references per instruction as possible; almost all instructions were able to contain at least one memory reference. Since the compactor was working under the same constraints, it could not be expected to perform significantly better than the hand-compaction. On the other hand, a poor compactor would have produced much worse results than the hand compaction, because poorly-scheduled operations would cost execution cycles; and later operations could not be squeezed tighter to make up for those lost cycles.

6.4 Performance Graphs

The graphs in Figures 6.1–6.8 plot execution time against various resource limitations. In all cases, execution time is measured as the number of "cycles" required to execute the benchmark program, where each instruction (i.e., group of operations) takes exactly one cycle. Each page includes two graphs; both graphs refer to the behavior of one benchmark. The vertical scale measures execution time as the total number of simulated execution cycles. The horizontal scale represents the various constraints placed on the processor's resources. These constraints are represented by groups of three or four numbers. As indicated along the graph's axis, in a group of three numbers, the first is the total number of ALUs (including address offset generators), the second is the number of buses, and the third is the memory bandwidth selected during compaction; in a group of four numbers (used for the automatically-compacted benchmarks), the first digit represents the number of address offset generators, the second is the number of ALUs (*not* including the address generators), and the third and fourth are the number of buses and memory bandwidth.

The top graph for a benchmark indicates how execution time is affected by the processor's bandwidth to memory. In this graph, the processor is assume to have an "infinite" number of ALUs and buses; the only limitations on concurrency are data dependencies, and the processor's bandwidth to memory. Note that this graph examines four different memory bandwidths: from left to right, unlimited bandwidth; quadruple-word bandwidth; double-word bandwidth; and single-word bandwidth.

The bottom graph for a benchmark analyzes the effect of the number of buses and ALUs on execution time, holding the memory bandwidth constant at double-word reads and writes (except for the tests with the fewest number of buses and ALUs, in which only single-word memory bandwidth can be effectively used).

The leftmost data point on this graph corresponds to having unlimited ALUs and buses, with double-word memory bandwidth. Moving to the right in this graph, the numbers of ALUs and buses available are steadily decreased. Note that each data point represents either having the same number of buses as ALUs, or having one more bus than the number of ALUs. This information was included in a single graph because the number of ALUs and the number of buses are so closely related. There cannot be fewer buses than ALUs, or some ALUs would not be able to save their results; but there should also not be far more buses than ALUs, because there are relatively few transfers (not using an ALU) that would benefit from the extra buses.

One oddity that might be noted is that, in several of the performance graphs, compaction with no resource constraints actually produced slightly slower code than compaction with some resources slightly constrained. This effect has no profound significance; it simply indicates that the compactor's prioritizing algorithm is not quite optimal, and in a few cases does not schedule the "best" operations into an instruction.

The table in Figure 6.9 lists some additional information related to these graphs. For each benchmark, it gives the total number of times each type of resource is used during the benchmark's execution. This information can be used to supplement the execution time graphs; for example, it is possible to determine how fast a benchmark could be executed (with various resource limitations), if there were *no* restrictions—not even data dependency restrictions—on operation ordering. This would give an "absolute" lower bound on a benchmark's execution time, and might help indicate how much of a benchmark's execution time is due to data dependencies, and how much is caused by the other limiting factors.

The following two subsections draw some conclusions from analyzing the two sets of graphs.

6.5 Memory Bandwidth

Examining the top graphs for the benchmarks reveals an interesting fact. For each benchmark, the ability to reference two words of memory simultaneously gives far better performance than if only one word of memory can be accessed. In the case of `naive reverse`, the double-word memory reference ability nearly halves the benchmark's execution time. In the hand-compiled benchmarks, increasing the bandwidth to memory further, even to the point of unlimiting the bandwidth, produces little additional improvement in execution time. However, the two automatically-compiled benchmarks show a significant performance improvement when the bandwidth is increased to quad-words, or is unlimited.

These results can be explained by considering the operations that are performed in these benchmarks. As previously described, lists are implemented as a linked list of two-element cells. Both `concat` and `naive reverse` make heavy use of list traversal algorithms. The results of these benchmarks indicate that it is often possible to read or write both halves of a list cell simultaneously. When reading, the pointer can be temporarily stored until the next list cell is to be read; when writing, the pointer to the next list cell can be written in preparation for creating that next list cell.

The `diff` benchmark also shows a great benefit from double-word memory accesses, indicating that structure references can also benefit from the higher memory bandwidth. In this benchmark, note that any structure will contain a two-word functor/arity pair in its body; it is logical that these two words could often be read or written together. Also, this benchmark uses many two-element structures—in these structures, the two arguments are in consecutive memory locations after the functor/arity pair, and both arguments can often be accessed simultaneously as well. Although this benchmark uses no structures with more than two arguments, it would appear that higher-arity structures would also benefit from double-word reads and writes, by being able to access various pairs of arguments simultaneously.

The automatically-compiled benchmarks show a significant performance improvement when bandwidth is increased from single-wide to double-wide, but also continue to show improvement as the bandwidth is increased further. An analysis of the simulation traces revealed that much of this memory traffic is being caused by saving the processor state in choice points, and restoring that state during goal failure. However, also recall that the intermediate-code translator always saves all of the processor's argument registers, even when the compiler can determine that only a few need to be saved. In fact, out of all 8 registers, most choice points in these benchmarks really only need to save 2 or 3 of them. Thus, these results seem to indicate, not that the bandwidth to memory should be increased to quadruple-words or more, but rather that intelligent choice-point creation (and not saving unnecessary processor state) can significantly reduce memory bandwidth and execution time.

Therefore, in general, the significant improvement in execution speed using a double-word wide data path to memory seems to warrant including such a wider memory path in a high-performance design. Following from the results of this analysis, a double-word path to memory was used in analyzing the effect of the number of buses and ALUs on performance, described in the next subsection.

6.6 Internal Resources

The lower graph for each benchmark indicates performance as a function of the number of internal buses and ALUs in the processor. Although there is some variation between benchmarks, most benchmarks exhibit a sharp "knee" in this graph; to the left of that knee, increasing the available resources does not significantly improve the execution time; but to the right of that knee, as resources are more limited, the execution time begins to rise sharply. For the hand-coded benchmarks, this knee occurs near the points marked "232" or "222"; that is, where two ALUs and two or three buses are available in the processor (and double-word bandwidth to memory is used). For the automatically-compiled benchmarks, this knee occurs near the point "122", which represents having one ALU, one address offset generator, two buses, and double-word bandwidth to memory.

Although the use of the ALUs in the hand-compiled benchmarks was not automatically determined, an examination of the compacted code shows that many instructions require an address offset to be generated. Therefore, one "ALU" in **concat**, **naive reverse**, and **diff** should be considered as an address offset generator. Taking this into account, we can see that almost all of the benchmarks suggest the following minimal internal processor configuration:

- One specialized address-generator, that can add the contents of a base register to a (positive or negative) offset, to obtain a memory address;
- One general-purpose ALU that can perform general arithmetic and logical operations; and
- Two or three write buses or write ports to the processor's registers and external memory.

At least for the benchmarks analyzed in this project, and the tools used to generate the program traces, additional hardware resources does not significantly improve the overall performance. However, limiting any of these resources further does markedly decrease performance.

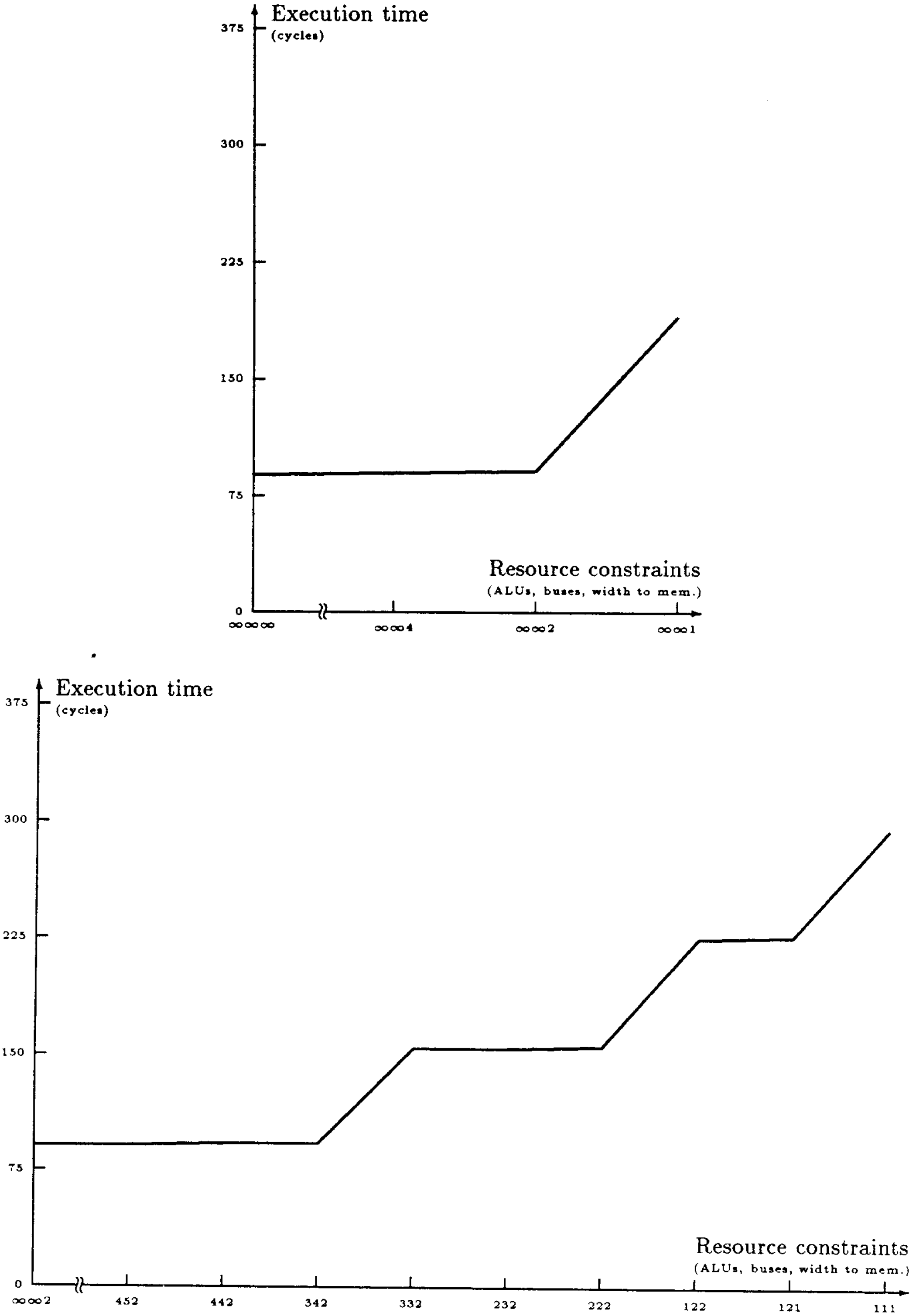


Figure 6.1: concat A Execution Times

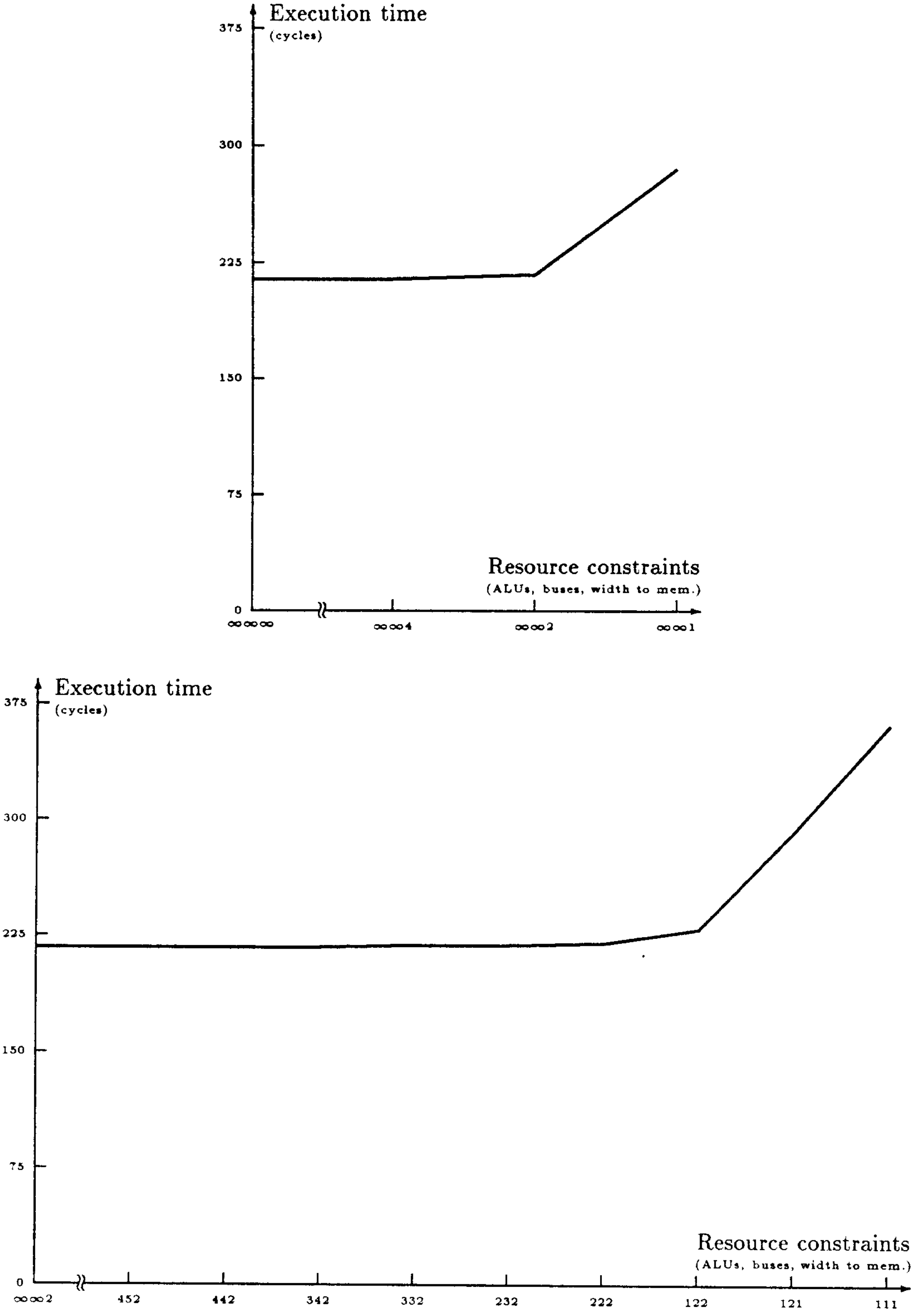


Figure 6.2: concat **B** Execution Times

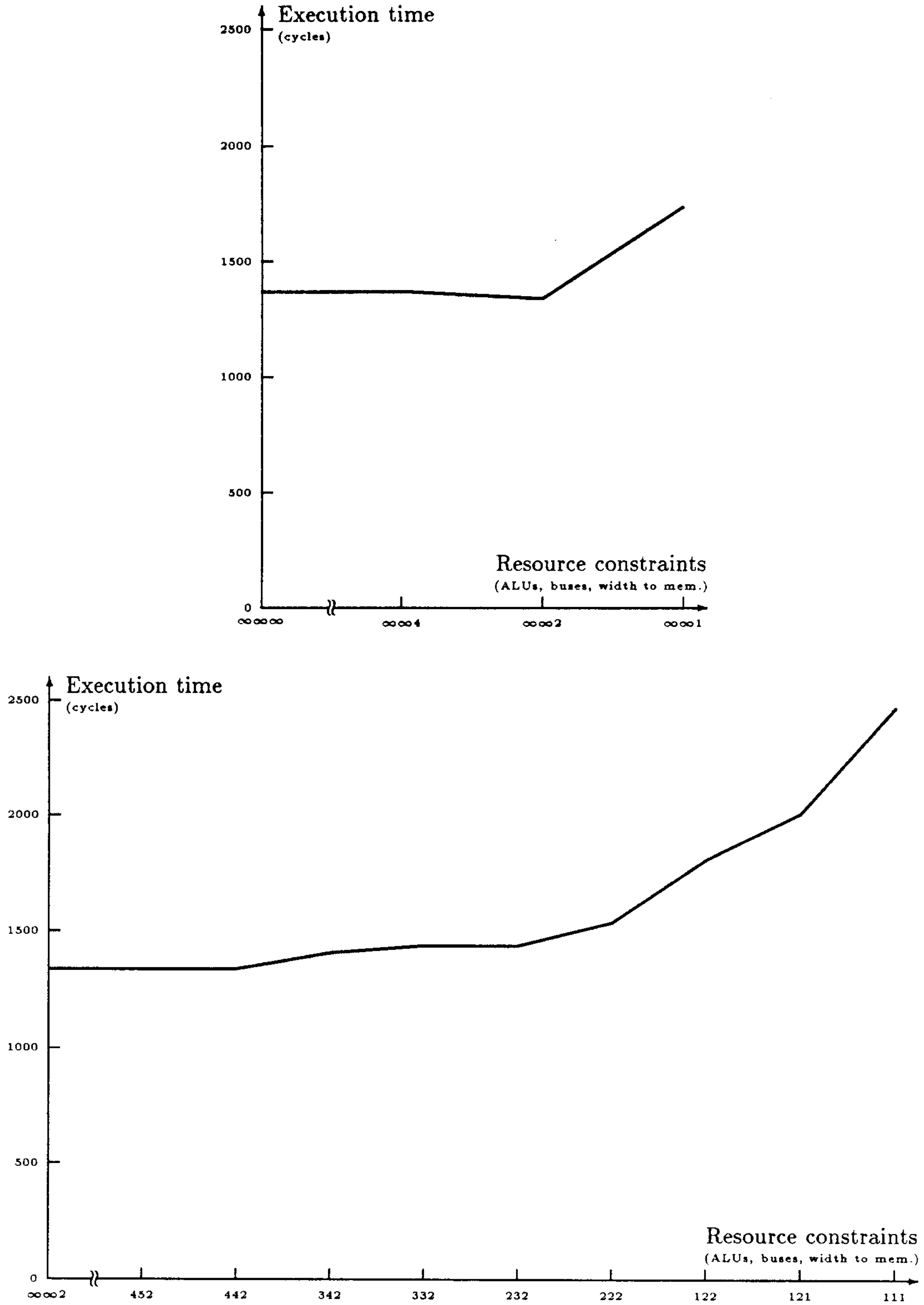


Figure 6.3: concat C Execution Times

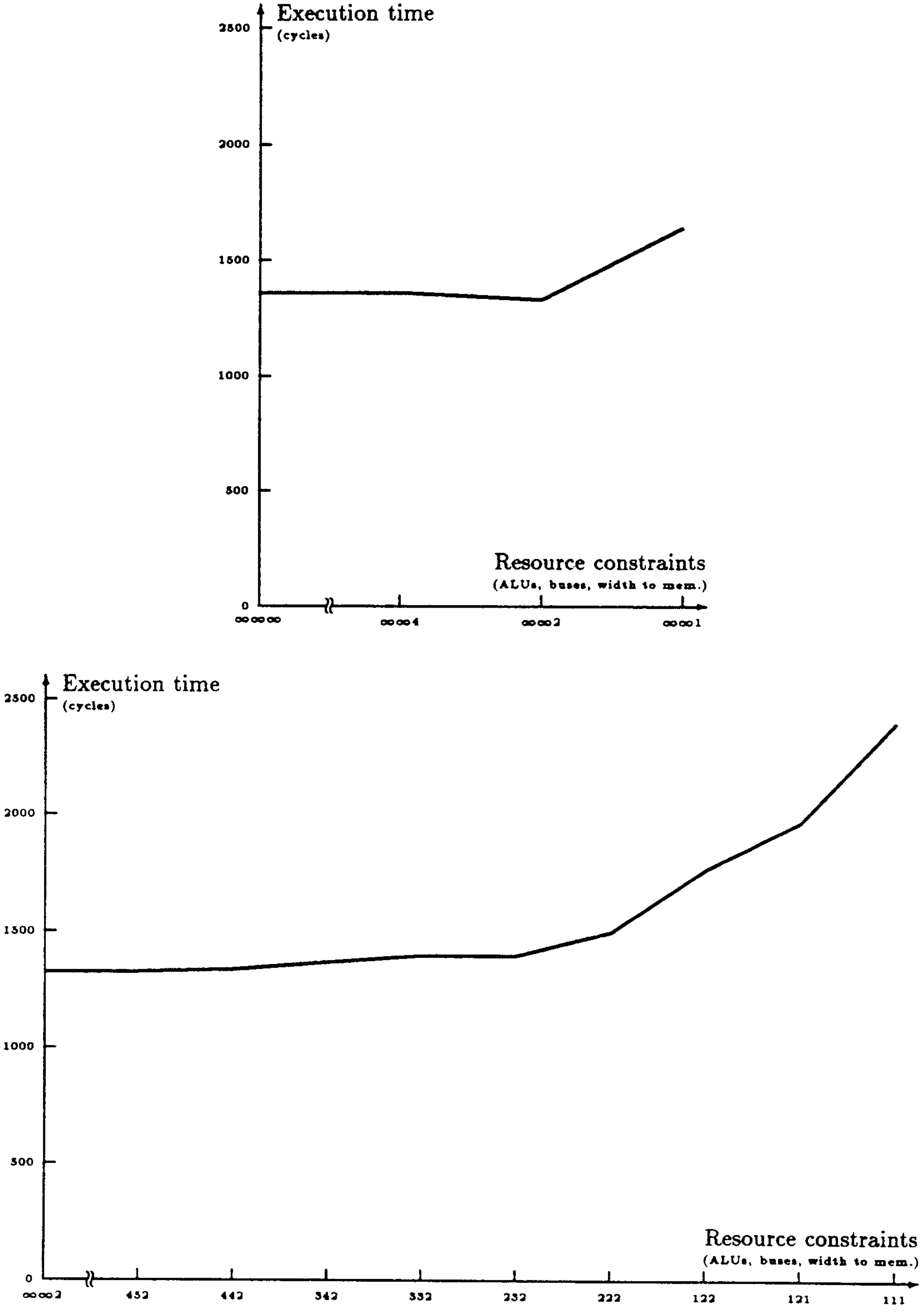


Figure 6.4: **concat D** Execution Times

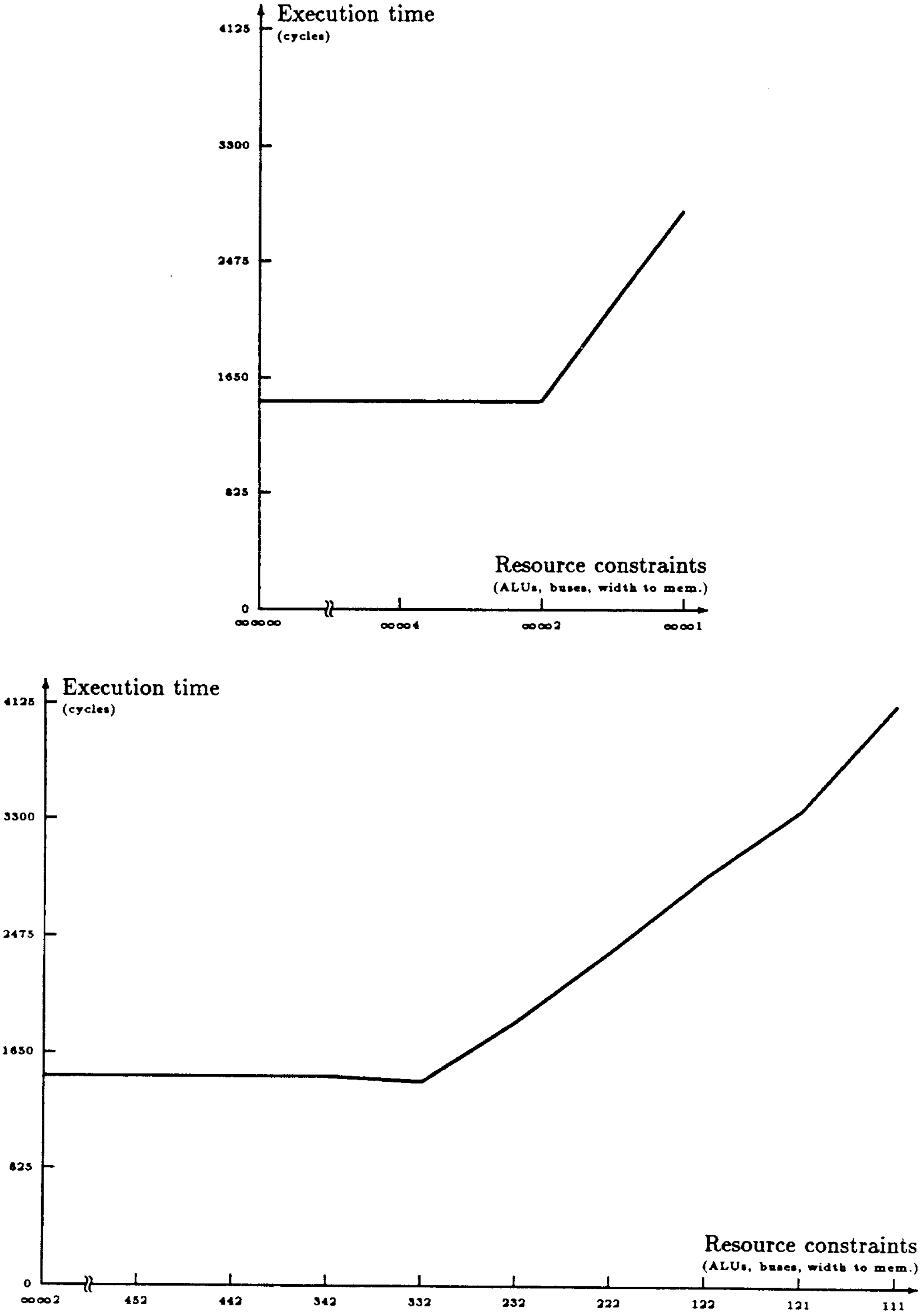


Figure 6.5: `naive reverse` Execution Times

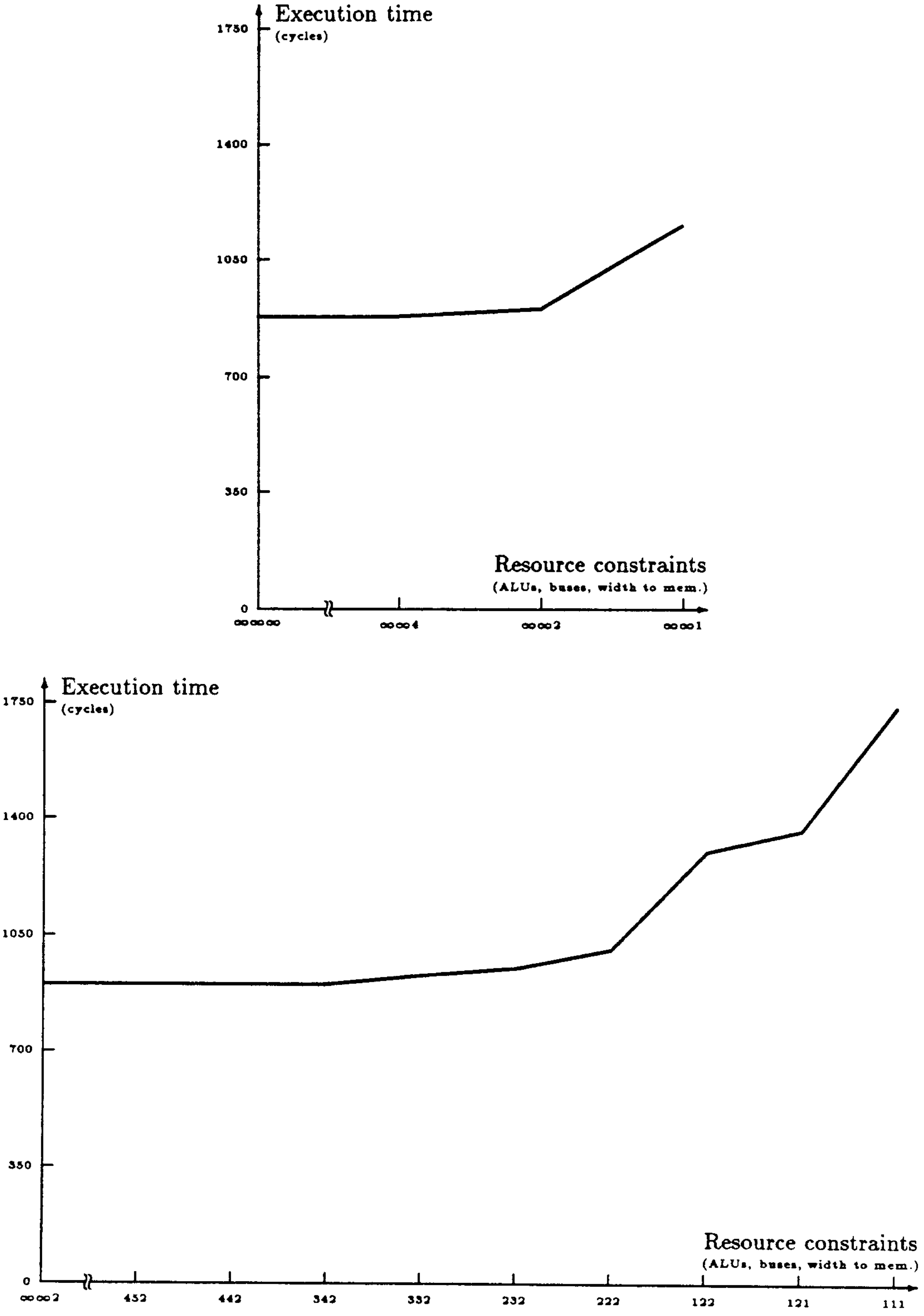


Figure 6.6: `diff` Execution Times

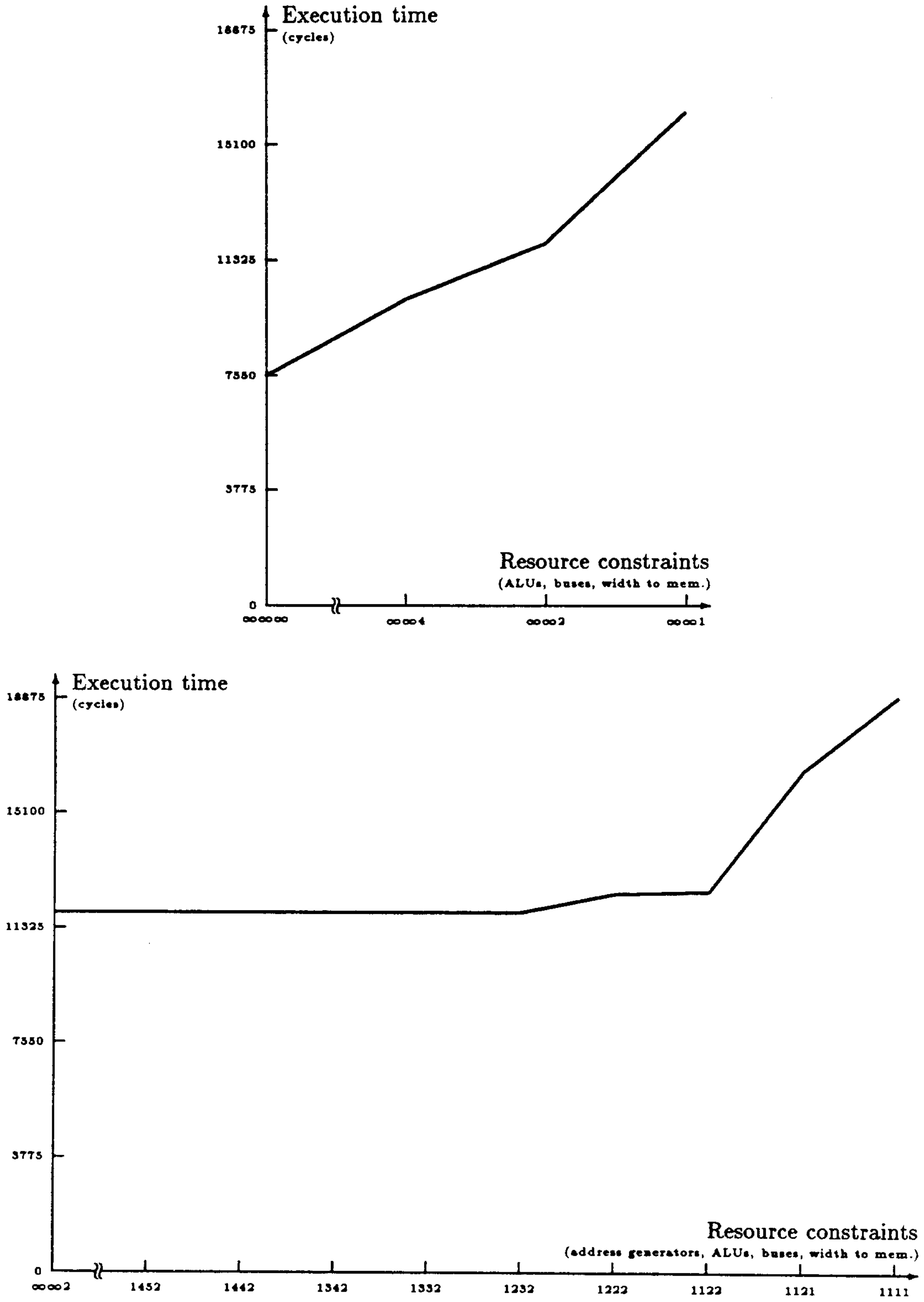


Figure 6.7: **four queens** Execution Times

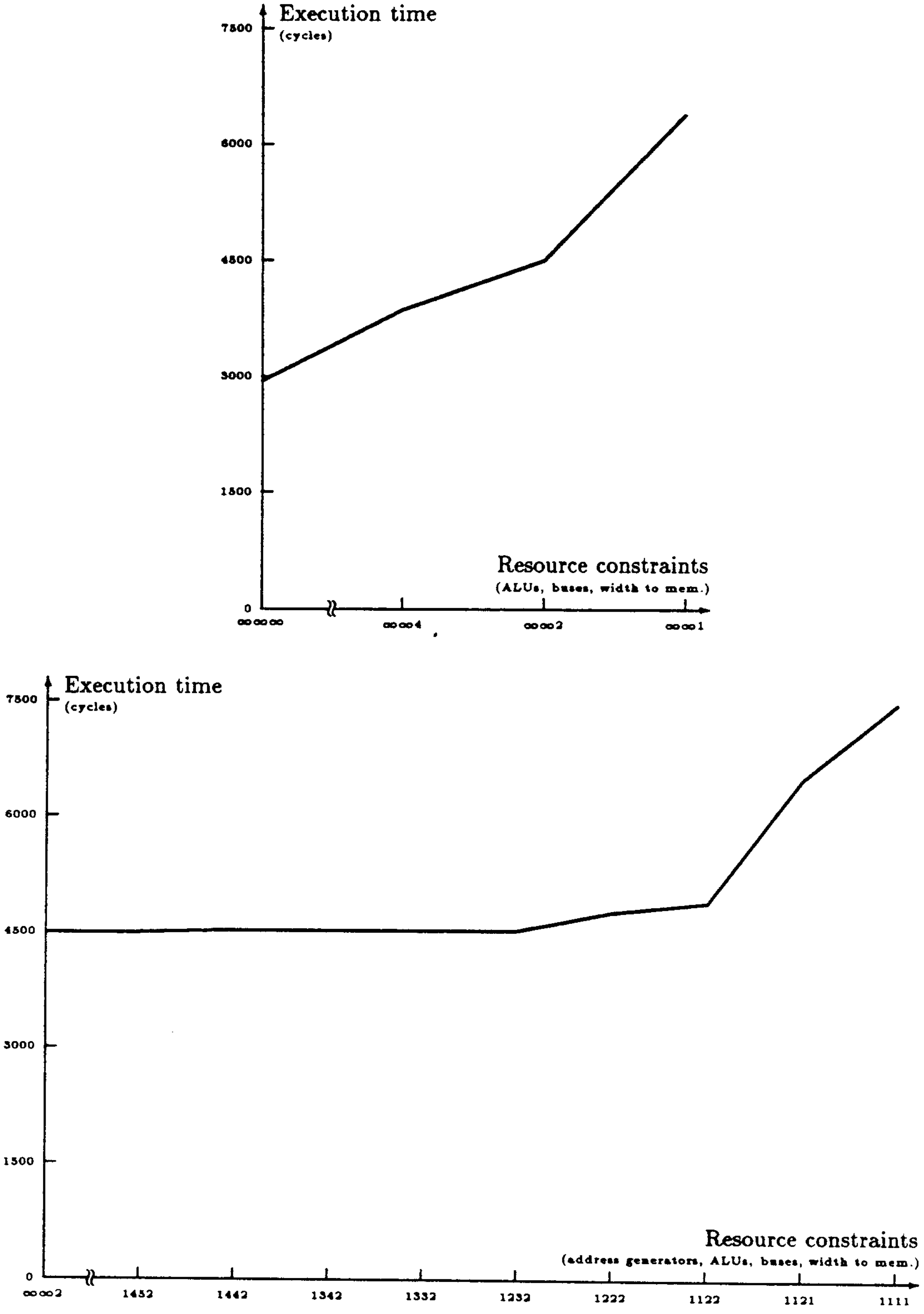


Figure 6.8: `mu math` Execution Times

Benchmark	Total number of times resource is used			
	ALUs	buses	memory (reads)	memory (writes)
`concat A`	216	257	74	77
`concat B`	190	231	136	47
`concat C`	1511	2035	705	598
`concat D`	1404	2093	729	600
`naive reverse`	2750	3549	1113	1142
`diff`	1247	1457	296	554
`four queens`	5044	15917	7548	5506
`mu math`	2175	6239	2966	2190

Figure 6.9: Total resource usage counts.

6.7 Possibilities for future work

Because of the limited scope of this project, there are of course many interesting tests and experiments that could not be performed. Perhaps the most important extension to this project would be the analysis of *large* compiled programs, to supplement the information gathered by this project for the smaller benchmarks. This extension would be more representative of real Prolog workloads, and would undoubtedly reveal additional requirements of a high-performance machine architecture. However, the current simulator executes fewer than 100,000 execution cycles per hour (on a Sun 3/50). To execute very large programs in a reasonable amount of time, the simulator would have to be rewritten (either in more efficient Prolog, or possibly in a different language), or it would have to be run on a much faster machine.

Also, only a limited set of resources were tracked by the tools in this project. There are almost limitless other types of resources that could be defined and tracked to help define a machine architecture. Some possible extensions would be more detailed analysis of separate data and address ALUs, or multiple special-function ALUs; buses that are restricted to communicate with only a subset of the internal registers, or separate read and write buses for registers; the tradeoffs of replacing registers by latches; and so on.

But even with the existing tools, more analyses could be performed. This could include live-register analysis of the simulation traces, to help determine how many registers can be effectively utilized; and frequency analysis of the instruction operations to help determine the types of instructions that can be effectively used.

7. Epilogue

In the process of writing the Prolog tools used in this project, I have of course learned much about Prolog (although, as with any language, it would take more than a couple of years to become a real "master" of the language). During this time, I have seen many things in the language and its current implementations that I liked; but I have also encountered things that made writing my programs difficult. This final section will briefly describe my observations.

First, it is clear that Prolog is well-suited to solving a wide range of problems, including (but not limited to) those that heavily involve searching and backtracking. The microcode compactor is a good example of this. A very small part of the compactor uses backtracking to help schedule operations—it is possible to schedule an operation into an instruction, and later find that doing so has prevented some other operation from *ever* being legally scheduled. The automatic backtracking of Prolog is invaluable for resolving such problems; for in a language like C, a programmer would have to explicitly write code to detect a poorly-scheduled operation, back up and undo all of the damage done by scheduling that operation, and then reschedule that operation and possibly many others.

But Prolog has benefits even in the bulk of the compactor's code, which is deterministic (does not do any backtracking): the binding of unbound variables allows information to propagate quickly to different parts of the program without the need for explicit "pointers". The use of unbound variables and Prolog's ability to dynamically allocate memory as it is needed—and in a manner invisible to the programmer—makes it easy to do things like gradually add operations to an instruction, without having to set an explicit and a priori upper limit on the number of operations per instruction.

On the other hand, several aspects of the Prolog language made writing the tools more difficult than I had hoped. One such aspect is the lack of a standard macro expansion or definition facility (in some cases, the ability to efficiently use global "variables" or even define global constants would have been sufficient). This lack created two problems in the compactor code: first, many predicates contain arguments that are passed, unchanged, from procedure to procedure, increasing the number of arguments that are needed—many predicates require around ten arguments. And second, I often found that changing the implementation of one procedure required rewriting many other procedures to supply it with the proper—and proper number of—arguments. This is not acceptable when writing very large programs.

Of course, there are many partial solutions to this problem. The "easy" solution would be to use Prolog's `assert` and `retract` to access global variables. Unfortunately, these operations are usually very slow. A second solution would be to run a separate macro expansion on a program before compiling it. The disadvantages of this approach are the inconvenience of the extra manual step, and the lack of a standard syntax. Another solution would be to encapsulate a procedure's arguments into one structure. Then, any change in the structure would be invisible to procedures that were simply passing the structure along. However, this one argument must still be passed (whereas it would not, if global variables were allowed); and, more seriously, any predicate that uses any part of the structure must first unwrap it. If such unwrapping is done individually by any predicate that needs to access a structure element, we still have the problem that any change to the structure will require changes to many different predicates; if such unwrapping is done by a common predicate, procedure call overhead can severely slow down program execution. Hopefully, as compilation techniques improve, this last objection can be mitigated or eliminated.

The second major problem I experienced is a problem, not so much with Prolog as a language, but with the current implementations. This problem is poor debugging support. The

Prolog implementations I have used provide very minimal debugging facilities—basically, the "four-port" model described by [CM81]. With this model (in which you can stop only when a procedure is called, exits, is retried, or fails), it is very difficult to examine or trace modifications of a variable, or to specify precise conditions or locations at which breakpoints should be placed. I often found the easiest way to debug a program was to insert `write` calls to print out the values of important variables at the appropriate locations in the program; in a language as advanced as Prolog, such primitive debugging methods should never be necessary.

A. Appendices

A.1 Intermediate Code Translator

```
/***************************************************************************
 *                  Intermediate Language-to-RTL Translator                  *
 *                        written by Richard Carlson                         *
 *                          version 1.1    5-18-89                           *
 ***************************************************************************/

/*     Copyright 1988,89  The Regents of the University of California      */
/*                          All Rights Reserved                             */


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This translator accepts the intermediate-language output of the Aquarius
% Prolog compiler, and translates it into RTL suitable for the microcode
% compactor.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This translator performs the following conversions:
%
%Add to all programs:
%+      generic fail subroutine
%+      general unify subroutine
%+
%Expressions:
%+      translate ^ into @
%+      translate [X] into mem(X)
%+      translate r(N) into rN
%+      translate b into cp     (different name for choice point register)
%+
%Operations:
%+      translate equal(X,Y,L) into goto([(X\==Y,L)])
%+      translate unify(X,Y,L) into preamble then jump to unify subroutine
%+      translate deref(X) into standard dereference loop
%+      translate trail(X) into standard trail code
%+      translate move(X,Y) into Y<--X
%+      translate jump(L) into goto([L])
%+      pass label(L) unchanged
%+
%+      translate switch(proc...) into multiway conditional branch
%+      try, retry, and trust translated into standard code
%+      translate test(ne,X,T,L) into goto([(X\?T,L)])
%+      translate test(eq,X,T,L) into goto([(X?T,L)])
%+      translate cmp({lt,le,gt,ge,eq,ne},X,Y) into
%+                              tf<--X{<,=<,>,>=,=,\=}Y
%+      translate jump({true,false},L) into goto([({tf,tf=0},L)])
%+      translate switch_on_term(...) into conditional goto
%+      hash stuff?
%+
%+      translate {add,sub,mul,div,and,or,not,lshift,rshift}(X,Y,Z)
%+              into Z <-- X {+,-,*,//,/\,\/,^,<<,>>} Y
%+
%+      translate procedure(N/A) into label(N/A)
%+      translate call(N/A, fixregs) into goto([(N/A)])
%+      allocate & deallocate translated into reg-munging instructions
%+      translate return into indirect goto
%+      translate fail into goto([fail])
%+      remove nop

:- op( 950, xfx, <--). % destructive assignment
:- op( 700, xfx, \=).  % test for value inequality
:- op( 700, xfx, ?).   % test for type equality
:- op( 700, xfx, \?).  % test for type inequality
:- op( 690, fy, ^).    % "pointer to"
:- op( 400, fy, ~).    % bitwise complement operator
:- op( 300, fx, ^^).   % "unbound variable"
:- op( 250, xfx, ^).   % concatenation of type and value fields (input)
:- op( 250, xfx, @).   % concatenation of type and value fields (output)


% Top-level call to translate a program.
%   arg 1: input:  name of the input file to translate
%     [the output is always written to the file 'xlate.ucode']
%
xlate(File) :-
        see(File),
        readlst(InList),
        seen,
        get_top_level_code(OutStart, OutRest),
        xlate(InList, OutRest),
        tell('xlate.ucode'),
        writelst(OutStart), write('.'), nl,
        told.

% Translate an input instruction list into an output RTL list.
%   arg 1: input:  a list containing the instructions to be translated
%   arg 2: output: a list containing the RTL equivalent instructions
%
xlate(In, Out) :-
        xlate(In, Out, 0).

% Same as xlate/2, but uses a third argument to generate unique labels
%   [args 1 and 2 same as above]
%   arg 3: input:  a numerical key that is updated as new labels are used
%
xlate([], CommonCode, _) :- !,
        get_common_code(CommonCode).
xlate([hash_length(Len) | InRest], OutRest, Key) :- !,
        xlatehash(Len, InRest, NewInRest, OutRest, NewOutRest, Key, NewKey),
        xlate(NewInRest, NewOutRest, NewKey).
xlate([In1 | InRest], OutRest, Key) :-
        xoper(In1, OutRest, NewOutRest, Key, NewKey),
        xlate(InRest, NewOutRest, NewKey).


% Translate a hash table into a sequence of conditional branches (not the
%       most efficient way to do this, but it works).
%   arg 1: input:  the number of hash pairs (left) in the hash table
%   arg 2: input:  input list of instructions, starting with hash table
%   arg 3: output: pointer into input list of instructions after table
%   arg 4: input:  output list of RTL code, starting with hash jumps
%   arg 5: output: pointer into output list of RTL code after jumps
%   arg 6: input:  key to be used to generate new labels
%   arg 7: output: new value of key that can be used to generate new labels
%
xlatehash(0, InRest, InRest, [goto([[hashfail]]) | Rest], Rest, Key, Key) :- !.
xlatehash(Num, [pair(Value,Label) | InRest], NewInRest,
                [goto([(hashreg=MyValue, Label), Key1]), label(Key1) | Rest],
                NewRest, Key, NewKey) :- !,
        xexpr(Value, MyValue),
        genkey(Key, TempKey, Key1),
        NewNum is Num - 1,
        xlatehash(NewNum, InRest, NewInRest, Rest, NewRest, TempKey, NewKey).
xlatehash(_Num, [BadOp | InRest], InRest, Rest, Rest, Key, Key) :-
        write('*** ERROR: *** bad op in hash table: '), write(BadOp), nl.

% Translate one intermediate instruction into (0 or more) RTL operations.
%   arg 1: input:  the next instruction to translate
%   arg 2: input:  the next location in the output RTL list
%   arg 3: output: location in the RTL list following this instruction's code
%   arg 4: input:  key to be used to generate new labels
%   arg 5: output: new value of key that can be used to generate new labels
%
xoper(equal(Src1,Src2,Lab), [goto([(MySrc1\==MySrc2,Lab), (Key1)]),
                  label(Key1) | Rest], Rest, Key, NewKey) :- !,
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2),
        genkey(Key, NewKey, Key1).
xoper(unify(Src1,Src2,_Lab), [(u1 <-- MySrc1), (u2 <-- MySrc2),
                  loadlabel(mem(ul),Key1), goto([unify]), label(Key1) | Rest],
                  Rest, Key, NewKey) :- !,
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2),
        genkey(Key, NewKey, Key1).
xoper(deref(r(Reg)), [goto([(type(MyReg)\?var, Key3), (Key1)]),
                  label(Key1), (tmp <-- MyReg), (MyReg <-- mem(MyReg)),
                  goto([((type(MyReg)?var), Key2), Key3]),
                  label(Key2),
                  goto([((MyReg\=tmp), Key1), Key3]),
                  label(Key3) | Rest], Rest, Key, NewKey) :- !,
        xexpr(r(Reg), MyReg),
        genkey(Key, NKey, Key1),
        genkey(NKey, NKey2, Key2),
        genkey(NKey2, NewKey, Key3).
xoper(deref(_Other), Rest, Rest, Key, Key) :- !.
xoper(trail(Addr), [(mem(tr) <-- MyAddr), (tr <-- tr+1) | Rest], Rest, Key, Key) :- !,
        xexpr(Addr, MyAddr).
xoper(move([Src],[Dest]), [(tempm <-- NewSrc), (NewDest <-- tempm) | Rest],
               Rest, Key, Key) :- !,
        xexpr([Src], NewSrc),
        xexpr([Dest], NewDest).
xoper(move(Src,Dest), [(NewDest <-- NewSrc) | Rest], Rest, Key, Key) :- !,
        xexpr(Src, NewSrc),
        xexpr(Dest, NewDest).
xoper(jump(Lab), [goto([Lab]) | Rest], Rest, Key, Key) :- !.
                        % (Make sure any label is preceded by a goto)
xoper(label(Lab), [goto([Lab]), label(Lab) | Rest], Rest, Key, Key) :- !.
xoper(switch(proc,Type,Expr,VarL,SpecL,FailL),
                [goto([(type(MyExpr)?var,VarL), (type(MyExpr)?MyType,SpecL),
                        (FailL)]) | Rest], Rest, Key, Key) :- !,
        xexpr(Expr, MyExpr),
        xtag(Type, MyType).
xoper(switch(unify,Type,Expr,VarCode,SpecCode,FailL), TotalCode, Rest, Key, NewKey) :- !,
        genkey(Key, Key1, VarL),
        genkey(Key1, Key2, SpecL),
        xoper(switch(proc,Type,Expr,VarL,SpecL,FailL), MyCode, LeftCode, Key2, NewKey),
        append([label(VarL) | VarCode], [label(SpecL) | SpecCode], LeftCode),
        append(MyCode, Rest, TotalCode).
%xoper(switch_on_term(Expr,VarL,LstL,AtmL,StrL),
%               [goto([(type(MyExpr)?tvar,VarL), (type(MyExpr)?tlst,LstL),
%                       (type(MyExpr)?tatm,AtmL), (type(MyExpr)?tstr,StrL),
%                       (fail)]) | Rest], Rest, Key, Key) :- !,
%       xexpr(Expr, MyExpr).
                % Note that, at least for now, *all* registers are saved
                % by the try/retry/trust instructions ("Regs" is ignored)
xoper(try(else,_Regs,Lab), [loadlabel(temp, Lab), loadlabel(temp2, Next),
                goto([try]), label(Next) | Rest], Rest, Key, NewKey) :- !,
        genkey(Key, NewKey, Next).
                        % Modify failure address in current choice point
xoper(retry(else,_Regs,Lab), [loadlabel(mem(cp+33), Lab) | Rest],
                Rest, Key, Key) :- !.
                        % Remove choice point & select previous one
xoper(trust(else,_Regs,fail), [(cp <-- mem(cp+36)), (s <-- s-38) | Rest],
                Rest, Key, Key) :- !.
xoper(test(ne,Src,Tag,Lab), [goto([(type(MySrc)\?MyTag,Lab), (Key1)]),
                label(Key1) | Rest], Rest, Key, NewKey) :- !,
        xexpr(Src, MySrc),
        xtag(Tag, MyTag),
        genkey(Key, NewKey, Key1).
xoper(test(eq,Src,Tag,Lab), [goto([(type(MySrc)?MyTag,Lab), (Key1)]),
                label(Key1) | Rest], Rest, Key, NewKey) :- !,
        xexpr(Src, MySrc),
        xtag(Tag, MyTag),
        genkey(Key, NewKey, Key1).
xoper(cmp(lt,Src1,Src2), [(tf <-- (MySrc1 < MySrc2)) | Rest], Rest, Key, Key) :- !,
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2).
xoper(cmp(le,Src1,Src2), [(tf <-- (MySrc1 =< MySrc2)) | Rest], Rest, Key, Key) :- !,
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2).
xoper(cmp(gt,Src1,Src2), [(tf <-- (MySrc1 > MySrc2)) | Rest], Rest, Key, Key) :- !,
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2).
xoper(cmp(ge,Src1,Src2), [(tf <-- (MySrc1 >= MySrc2)) | Rest], Rest, Key, Key) :- !,
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2).
xoper(cmp(eq,Src1,Src2), [(tf <-- (MySrc1 = MySrc2)) | Rest], Rest, Key, Key) :- !,
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2).
xoper(cmp(ne,Src1,Src2), [(tf <-- (MySrc1 \= MySrc2)) | Rest], Rest, Key, Key) :- !,
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2).
xoper(jump(true,Lab), [goto([(tf=1,Lab), (Key1)]), label(Key1) | Rest],
                    Rest, Key, NewKey) :- !,
        genkey(Key, NewKey, Key1).
xoper(jump(false,Lab), [goto([(tf=0,Lab), (Key1)]), label(Key1) | Rest],
                    Rest, Key, NewKey) :- !,
        genkey(Key, NewKey, Key1).
xoper(hash(Tag,Value,_Num,Table), [(hashreg <-- MyValue),
                         loadlabel(hashfail,Fail),
                         goto([(type(MyValue)?MyTag, Table), Fail]),
                         label(Fail) | Rest], Rest, Key, NewKey) :- !,
        xexpr(Value, MyValue),
        xtag(Tag, MyTag),
        genkey(Key, NewKey, Fail).

xoper(add(Src1,Src2,Dest), Code, Rest, Key, NewKey) :- !,
        xoper(deref(Src1), Code, Code1, Key, Key1),
        xoper(deref(Src2), Code1, Code2, Key1, NewKey),
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2),
        xexpr(Dest, MyDest),
        Code2 = [(MyDest <-- MySrc1 + MySrc2) | Rest].
xoper(sub(Src1,Src2,Dest), Code, Rest, Key, NewKey) :- !,
        xoper(deref(Src1), Code, Code1, Key, Key1),
        xoper(deref(Src2), Code1, Code2, Key1, NewKey),
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2),
        xexpr(Dest, MyDest),
        Code2 = [(MyDest <-- MySrc1 - MySrc2) | Rest].
%xoper(mul(Src1,Src2,Dest), [(MyDest <-- MySrc1 * MySrc2) | Rest], Rest, Key, Key) :- !,
%       xexpr(Src1, MySrc1),
%       xexpr(Src2, MySrc2),
%       xexpr(Dest, MyDest).
%xoper(div(Src1,Src2,Dest), [(MyDest <-- MySrc1 // MySrc2) | Rest], Rest, Key, Key) :- !,
%       xexpr(Src1, MySrc1),
%       xexpr(Src2, MySrc2),
%       xexpr(Dest, MyDest).
xoper(and(Src1,Src2,Dest), Code, Rest, Key, NewKey) :- !,
        xoper(deref(Src1), Code, Code1, Key, Key1),
        xoper(deref(Src2), Code1, Code2, Key1, NewKey),
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2),
        xexpr(Dest, MyDest),
        Code2 = [(MyDest <-- MySrc1 /\ MySrc2) | Rest].
xoper(or(Src1,Src2,Dest), Code, Rest, Key, NewKey) :- !,
        xoper(deref(Src1), Code, Code1, Key, Key1),
        xoper(deref(Src2), Code1, Code2, Key1, NewKey),
        xexpr(Src1, MySrc1),
        xexpr(Src2, MySrc2),
        xexpr(Dest, MyDest),
        Code2 = [(MyDest <-- MySrc1 \/ MySrc2) | Rest].
xoper(not(Src,Dest), Code, Rest, Key, NewKey) :- !,
        xoper(deref(Src), Code, Code1, Key, NewKey),
        xexpr(Src, MySrc),
        xexpr(Dest, MyDest),
        Code1 = [(MyDest <-- ~ MySrc) | Rest].
%xoper(lshift(Src1,Src2,Dest), [(MyDest <-- MySrc1 << MySrc2) | Rest],
%                        Rest, Key, Key) :- !,
%       xexpr(Src1, MySrc1),
%       xexpr(Src2, MySrc2),
%       xexpr(Dest, MyDest).
%xoper(rshift(Src1,Src2,Dest), [(MyDest <-- MySrc1 >> MySrc2) | Rest],
%                        Rest, Key, Key) :- !,
%       xexpr(Src1, MySrc1),
%       xexpr(Src2, MySrc2),
%       xexpr(Dest, MyDest).

xoper(procedure(Lab), [label(Lab) | Rest], Rest, Key, Key) :- !.
                         % Call new procedure; set return address
xoper(call(Dest,fixregs), [loadlabel(retaddr,Return), goto([Dest]),
                            label(Return) | Rest], Rest, Key, NewKey) :- !,
        genkey(Key, NewKey, Return).
xoper(allocate(Num), [(mem(s+EOffset) <-- e), (mem(s+RetOffset) <-- retaddr),
                (e <-- s), (s <-- s+FrameSize) | Rest], Rest, Key, Key) :- !,
        FrameSize is ((Num + 1)\/1) + 1,          % Add 2 or 3 to make even
        EOffset is FrameSize - 2,
        RetOffset is FrameSize - 1.
xoper(deallocate(Num), [(retaddr <-- mem(e+RetOffset)),
                (e <-- mem(e+EOffset)) | Rest], Rest, Key, Key) :- !,
        FrameSize is ((Num + 1)\/1) + 1,          % Add 2 or 3 to make even
        EOffset is FrameSize - 2,
        RetOffset is FrameSize - 1.
xoper(return, [goto([[retaddr]]) | Rest], Rest, Key, Key) :- !.
xoper(fail, [goto([fail]) | Rest], Rest, Key, Key) :- !.
xoper(nop, Rest, Rest, Key, Key) :- !.

xoper(Other, Rest, Rest, Key, Key) :-
        write('*** ERROR: instruction not translated: '), write(Other), nl.


% Translate an expression into a form acceptable to the compactor.
%   arg 1: input:  an expression from the input intermediate language list
%   arg 2: output: a translated equivalent form of that expression
%
xexpr((Tag^Value), (MyTag@NewValue)) :- !,
        xtag(Tag, MyTag),
        xexpr(Value, NewValue).
xexpr([], nil) :- !.
xexpr([Addr], mem(NewAddr)) :- !,
        xexpr(Addr, NewAddr).
xexpr(r(Num), NewReg) :- !,
        MyNum is Num + 1,
        name(MyNum, NumName),
        append("r", NumName, RegName),
        name(NewReg, RegName).
xexpr(Unary, NewUnary) :-
        Unary =.. [Op, Arg], !,
        xexpr(Arg, NewArg),
        NewUnary =.. [Op, NewArg].
xexpr(Binary, NewBinary) :-
        Binary =.. [Op, Arg1, Arg2], !,
        xexpr(Arg1, NewArg1),
        xexpr(Arg2, NewArg2),
        NewBinary =.. [Op, NewArg1, NewArg2].
```

```
xexpr(b, cp) :- !.
xexpr(Other, Other).

% Translate a tag expression into a form acceptable to the compactor.
%   arg 1: input:  a tag name from the input intermediate language list
%   arg 2: output: a translated equivalent form of that tag
%
xtag(tstr, struct) :- !.
xtag(tlst, list) :- !.
xtag(tatm, const) :- !.
xtag(tvar, var) :- !.
xtag(BadTag, var) :-
        write('*** ERROR: unknown tag: '), write(BadTag), nl.


% Generate a new unique label name from a given key, and update that key.
%   arg 1: input:  the current value of the numeric key
%   arg 2: output: a new value for that key
%   arg 3: output: a new, unique label name
%
genkey(Key, NewKey, KeySym) :-
        name(Key, KeyStr),
        append("$key_", KeyStr, KeyName),
        name(KeySym, KeyName),
        NewKey is Key + 1.


% Get ''top-level'' code that should begin any benchmark program.
%   arg 1: output: RTL list of the top-level code
%
get_top_level_code([
        set([h= ^mem(1000), s= ^mem(30000), tr= ^mem(60000), ul= ^mem(90000)]) ,
        mem(tr) <-- const@0 ,           % Create "dummy space" on trail
        mem(tr+1) <-- const@0 ,
        tr <-- tr+2 ,
                                % Set any initial data structures in memory
        set([mem(99)=[1,2,3],mem(179)=[4,5,6]]) ,
    {[], [newh], []},
        set([mem(h)={user},mem(h+1)={user},mem(h+2)={user},mem(h+3)={user},
             mem(h+4)={user},mem(h+5)={user},mem(h+6)={user},mem(h+7)={user}]),
        r1 <-- var@h ,
        r2 <-- var@h+1 ,
        r3 <-- var@h+2 ,
        r4 <-- var@h+3 ,
        r5 <-- var@h+4 ,
        r6 <-- var@h+5 ,
        r7 <-- var@h+6 ,
        r8 <-- var@h+7 ,
        origh <-- h ,
    {newh},
        h <-- h+8 ,
        cp <-- 0 ,
        e <-- 0 ,
        goto([main]) ,
    label(main) ,
        loadlabel(temp, mainfail) ,     % Create choice point
        loadlabel(temp2, mainfirst) ,
        goto([try]) ,
    label(mainfirst) ,          % First clause of "main": call the benchmark
        loadlabel(retaddr, mainret) ,   % Establish return address
        goto([benchcode]) ,
    label(mainret) ,
        show([mem(origh), mem(origh+1), mem(origh+2), mem(origh+3),
              mem(origh+4), mem(origh+5), mem(origh+6), mem(origh+7)]) ,
        goto([fail]) ,                          % Try to find other solutions
    label(mainfail) ,           % Second clause of "main": benchmark failed
        show([{'no (more) solutions'}]) ,
        end ,
    label(benchcode)    | Rest], Rest).


% Get common code that can be used as subroutines for very long translations.
%   arg 1: output: RTL list of the common subroutine code
%
get_common_code([
    label(try) ,        % try code: create a choice point
        mem(s+0) <-- r1,                % Save argument registers
        mem(s+1) <-- r2,
        mem(s+2) <-- r3,
        mem(s+3) <-- r4,
        mem(s+4) <-- r5,
        mem(s+5) <-- r6,
        mem(s+6) <-- r7,
        mem(s+7) <-- r8,
%       mem(s+8) <-- r9,                % For now, processor has only 8 regs.
%       mem(s+9) <-- r10,
%       mem(s+10) <-- r11,
%       mem(s+11) <-- r12,
%       mem(s+12) <-- r13,
%       mem(s+13) <-- r14,
%       mem(s+14) <-- r15,
%       mem(s+15) <-- r16,
%       mem(s+16) <-- r17,
%       mem(s+17) <-- r18,
%       mem(s+18) <-- r19,
%       mem(s+19) <-- r20,
%       mem(s+20) <-- r21,
%       mem(s+21) <-- r22,
%       mem(s+22) <-- r23,
%       mem(s+23) <-- r24,
%       mem(s+24) <-- r25,
%       mem(s+25) <-- r26,
%       mem(s+26) <-- r27,
%       mem(s+27) <-- r28,
%       mem(s+28) <-- r29,
%       mem(s+29) <-- r30,
%       mem(s+30) <-- r31,
%       mem(s+31) <-- r32,
        mem(s+31) <-- retaddr ,         % Save current return address
        mem(s+32) <-- fp ,              % Save current frame pointer
        mem(s+33) <-- temp ,            % Save alternate (failure) code address
        temp3 <-- tr-1 ,                % Make sure trail is on even boundary
    goto([((tr /\ 1), trodd),
          (treven)]) ,
    label(trodd) ,
    temp <-- mem(tr-1) ,
    mem(tr) <-- temp ,
    tr <-- tr+1 ,
    goto([treven]) ,
    label(treven) ,
    mem(s+34) <-- h ,               % Save current heap pointer
    mem(s+35) <-- tr ,              % Save current trail pointer
    mem(s+36) <-- cp ,              % Save previous choice point
    mem(s+37) <-- e ,
    cp <-- s ,              % Update CP to point to this choice point
    s <-- s+38 ,
    goto([[temp2]]) ,               % Jump to first code choice

label(unify) ,
    uret <-- mem(ul) ,
    goto([(type(u1) ? const, unic1),
          (type(u1) ? list, unil1),
          (type(u1) ? struct, unis1),
          (univ1)]) ,

    label(univ1) ,
    tmpu <-- u1 ,
    u1 <-- mem(u1) ,
    goto([((u1==tmpu), u1varnottmpu),       % u1 is unbound
          (univ1a)]) ,
    label(univ1a) ,
    goto([((type(u1) ? const), unic1),
          ((type(u1) ? list), unil1),
          ((type(u1) ? struct), unis1),
          (univ1)]) ,

    label(u1varnottmpu) ,
    goto([((u1 == u2), uuret),
          (u1notequ2)]) ,
    label(u1notequ2) ,
    goto([((type(u2) \? var), univ1x2),
          (univ1v2)]) ,
    label(univ1v2) ,
     tmpu <-- u2 ,
     u2 <-- mem(u2) ,
    goto([((u2==tmpu), univ1unb2),
          (univ1v2a)]) ,
    label(univ1v2a) ,
    goto([((type(u2) ? var), univ1v2),
          (univ1x2)]) ,
    label(univ1unb2) ,
    goto([((u1=u2), uuret), % done if same variables
          (univ1nev2)]) ,
    label(univ1nev2) ,
    goto([((u1 > u2), univ1x2),
          (unix1v2)]) ,     % u1 < u2

    label(univ1x2) ,
mem(u1) <-- u2 ,
mem(tr) <-- u1 ,
tr <-- tr+1 ,
goto([[uret]]) ,

label(unix1v2) ,
mem(u2) <-- u1 ,
mem(tr) <-- u2 ,
tr <-- tr+1 ,
goto([[uret]]) ,

label(unic1) ,
goto([((type(u2) ? const), unic1c2),
      ((type(u2) ? list), fail),
      ((type(u2) ? struct), fail),
      (unic1v2)]) ,
label(unic1v2) ,
tmpu <-- u2 ,
u2 <-- mem(u2) ,
goto([((u2==tmpu), unix1v2),
      (unic1v2a)]) ,
label(unic1v2a) ,
goto([((type(u2) ? const), unic1c2),
      ((type(u2) ? struct), fail),
      ((type(u2) ? list), fail),
      (unic1v2)]) ,

label(unic1c2) ,
goto([(u1 == u2, uuret),
      (fail)]) ,

label(unis1) ,
goto([((type(u2) ? struct), unis1s2),
      ((type(u2) ? const), fail),
      ((type(u2) ? list), fail),
      (unis1v2)]) ,
label(unis1v2) ,
tmpu <-- u2 ,
u2 <-- mem(u2) ,
goto([((u2==tmpu), unix1v2),
      (unis1v2a)]) ,
label(unis1v2a) ,
goto([((type(u2) ? struct), unis1s2),
      ((type(u2) ? const), fail),
      ((type(u2) ? list), fail),
      (unis1v2)]) ,

label(unis1s2) ,
goto([(u1 == u2, uuret),
      (fail)]) ,        % would need more code to unify structures

label(unil1) ,
goto([(type(u2) ? list, unil1l2),
      (type(u2) ? const, fail),
      (type(u2) ? struct, fail),
      (unil1v2)]) ,
        label(unil1v2) ,
        tmpu <-- u2 ,
        u2 <-- mem(u2) ,
        goto([((u2==tmpu), unix1v2),
              (unil1v2a)]) ,
        label(unil1v2a) ,
        goto([((type(u2) ? list), unil1l2),
              ((type(u2) ? const), fail),
              ((type(u2) ? struct), fail),
              (unil1v2)]) ,

        label(unil1l2) ,
        goto([(u1 == u2, uuret),
              (unil1l2a)]) ,
        label(unil1l2a) ,
        head1 <-- mem(u1) ,
        tail1 <-- mem(u1+1) ,
        head2 <-- mem(u2) ,
        tail2 <-- mem(u2+1) ,
        mem(ul+2) <-- tail1 ,
        mem(ul+3) <-- tail2 ,
        ul <-- ul+4 ,
        goto([(head1 == head2, uniltail),
              (unil1l2b)]) ,
        label(unil1l2b) ,
        loadlabel(mem(ul), uniltail) ,
        u1 <-- head1 ,
        u2 <-- head2 ,
        goto([unify]) ,
        label(uniltail) ,
        u1 <-- mem(ul-2) ,
        u2 <-- mem(ul-1) ,
        ul <-- ul-4 ,
        goto([unify]) ,

        label(uuret) ,
        goto([[uret]]) ,

    label(fail) ,
        r1 <-- mem(cp+0) ,             % Restore argument registers
        r2 <-- mem(cp+1) ,
        r3 <-- mem(cp+2) ,
        r4 <-- mem(cp+3) ,
        r5 <-- mem(cp+4) ,
        r6 <-- mem(cp+5) ,
        r7 <-- mem(cp+6) ,
        r8 <-- mem(cp+7) ,
%       r9 <-- mem(cp+8) ,
%       r10 <-- mem(cp+9) ,
%       r11 <-- mem(cp+10) ,
%       r12 <-- mem(cp+11) ,
%       r13 <-- mem(cp+12) ,
%       r14 <-- mem(cp+13) ,
%       r15 <-- mem(cp+14) ,
%       r16 <-- mem(cp+15) ,
%       r17 <-- mem(cp+16) ,
%       r18 <-- mem(cp+17) ,
%       r19 <-- mem(cp+18) ,
%       r20 <-- mem(cp+19) ,
%       r21 <-- mem(cp+20) ,
%       r22 <-- mem(cp+21) ,
%       r23 <-- mem(cp+22) ,
%       r24 <-- mem(cp+23) ,
%       r25 <-- mem(cp+24) ,
%       r26 <-- mem(cp+25) ,
%       r27 <-- mem(cp+26) ,
%       r28 <-- mem(cp+27) ,
%       r29 <-- mem(cp+28) ,
%       r30 <-- mem(cp+29) ,
%       r31 <-- mem(cp+30) ,
%       r32 <-- mem(cp+31) ,
        retaddr <-- mem(cp+31) ,        % Get back previous return address
        fp <-- mem(cp+32) ,             % Return to previous stack frame
        temp <-- mem(cp+33) ,           % Get failure code address
        h <-- mem(cp+34) ,              % Reclaim heap space
        newtr <-- mem(cp+35) ,          % Get previous trail pointer
        e <-- mem(cp+37) ,              % Restore previous environment frame
        goto([((tr /\ 1), failodd),     % Reclaim & unbind trail
              (failloop)]) ,
        label(failloop) ,
        tr1 <-- mem(tr-1) ,
        tr2 <-- mem(tr-2) ,
        goto([((tr = newtr), newfail),
              (failcont)]) ,
        label(failcont) ,
        mem(tr1) <-- var@tr1 ,
        mem(tr2) <-- var@tr2 ,
        tr <-- tr-2 ,
        goto([failloop]) ,
        label(failodd) ,
        tr <-- tr-1 ,
        tr1 <-- mem(tr) ,
        mem(tr1) <-- var@tr1 ,
        goto([failloop]) ,
        label(newfail) ,
        goto([[temp]])
]).


%%%%%%%%%%%%%%%%%%%%%%%%%%% Miscellaneous subroutines %%%%%%%%%%%%%%%%%%%%%%%%%%

% The ubiquitous list append/concat routine.
%   arg 3 is the concatenation of arg 1 and arg 2
%
append([], List, List) :- !.
append([First | Rest], List, [First | New]) :-
        append(Rest, List, New).


% Read a sequence of Prolog facts and assemble them into a list.
%   arg 1: output: list of (intermediate instructions) from current input
%
```

```
readlst(List) :-
        read(Next),
        (Next == end_of_file ->
                List = [] ;
                (List = [Next | Rest],
                 readlst(Rest)
                )
        ).


% Pretty print a (possibly nested) list, one (indented) element per line.
%   arg 1: input:  list of (RTL) operations to be printed
%
writelst(List) :-
        writelst(List, 0).

%   arg 1: input:  list of operations to be printed
%   arg 2: input:  current level of indentation
%
writelst([First | Rest], Num) :-
        !,
        tab(Num), write('['), nl,
        NewNum is Num+1,
        writesub([First | Rest], NewNum).
writelst(NonList, Num) :-
        tab(Num), writeq(NonList).

writesub([], Num) :- !,
        NewNum is Num-1,
        tab(NewNum), write(']').
writesub([Only], Num) :- !,
        writelst(Only, Num), nl,
        writesub([], Num).
writesub([First | Rest], Num) :-
        writelst(First, Num), write(','), nl,
        writesub(Rest, Num).
```

A.2 Microcode Compactor

A.2.1 Main Compactor

```
/*************************************************************************
 *                       Microoperation Compactor                        *
 *                      written by Richard Carlson                       *
 *                        version 2.1    5-18-89                         *
 *************************************************************************/

/*      Copyright 1988,89  The Regents of the University of California  */
/*                         All Rights Reserved                          */


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This compactor accepts a seqence of primitive microoperations, determines
% data dependencies between those operations, and then packs these operations
% into an operationally-equivalent list of microinstructions (each of which
% may contain several microoperations).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Operation:
%  "compact(File)"
%    reads input microcode list from the file "File.ucode"
%    writes output compacted list to the file "File.compact"
%
% Format of the input microcode:
%  A single list, containing
%    primitive microoperations, and
%    user-specified constraint information
%
% Format of the output compacted code:
%  A "{}" structure, containing resource tracking information, then
%  A list of "instructions", where each instruction is a list of:
%    0 or more operations, followed by one "{}" resource usage structure
%
% Possible microoperations (both input and output):
%  label(Label)         Creates a destination point for a goto operation
%  goto(DestList)       Changes program flow (conditionally or uncond.).
%                       Each element of the DestList can be either
%                         a simple Label, for an unconditional branch; or
%                         (Source,Label), for a branch only if Source is true;
%                         see the list of legal conditions below.
%  Dest <-- Source      Transfers the contents of Source into Dest.
%                       See the list below for legal Source and Dest values.
%  set([Dest=Source,..])"Free" transfer of Source into Dest; i.e., no
%                         resources are used by this operation.  Used to
%                         initialize constants, etc.
%                       Source can include lists, structures, ^'s to indicate
%                         dereferencing, ^^'s to indicate unbound variables,
%                         and {user} to read the source from the terminal.
%                       "set"s and "show"s are never re-ordered.
%  show([Dest,...])     Prints the current value of Dest on the terminal.
%                       Dest can also be {Atom}, where the Atom itself is
%                         printed (instead of trying to look up its value).
%                       "set"s and "show"s are never re-ordered.
%  end                  Indicates the end of a program's execution.
%
% Pseudo-operations for specifying manual constraints (in the input):
%  {Marker}             Marks the following operation so that manual
%                       constraints can be placed on its scheduling.
%  {CoReq,PreReq,Forced} Lists other operations that have manual constraints
%                       from the operation immediately following this one.
%                       CoReq and PreReq list Markers of operations that
%                       must be scheduled at the same time or later, and
%                       operations that must be scheduled later than this one.
%                       A Forced element has the form (Marker,Delay), and
%                       forces the marked operation to be scheduled exactly
%                       Delay instructions after this one.
%
% Resource tracking information (in the output):
%  The initial tracking information structure has the form
%    {n, Res1Name, Res2Name, ... , ResnName}
%    where "n" is the number of different resources being tracked; and
%    each "ResName" is a symbolic name describing a tracked resource
%  The resource usage structure in each instruction has the form
%    {Res1Value, Res2Value, ... , ResnValue}
%    where each "ResValue" indicates the use of the corresponding resource
%
% Legal "Source" values:
%  constant value       specifies a literal value to transfer
%  tag@value            specifies a literal tag and value
%  atom                 specifies a register (or other fixed location)
%  mem(Address)         specifies a memory location -- Address can
%                       specify an offset from any base register
%  +,-                  perform addition or subtraction
%  /\,\/,~              perform logical and, or, negation
%  ==,\==               compare both tag and value      \ Comparison operations;
%  ?,\?                 compare only tags                >evaluate to 1 if true,
%  =,\=,<,>,<=,>=       compare only values             / or 0 if false.
%
% Legal "Dest" values:
%  atom                 specifies a register (or other fixed location)
%  mem(Address)         specifies a memory location -- Address can
%                       specify an offset from any base register
%
% The resources that are currently tracked and reported on are:
%  alu:         the number of arithmetic operations used
%  bus:         the number of buses (transfers) used
%  memread:     the number of reads performed from memory
%  memwrite:    the number of writes performed to memory
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%% Implementation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% structure of an uncompacted operation node:
%  node(OP,     % 1     % The actual code/operation to perform
%       BLK,    % 2     % Block number for this operation
%       IDX,    % 3     % Unique index within block for this operation
%       TYP,    % 4     % Type of node this is:
```

```
%                              1 = standard trace node
%                              0 = artificial label node
%                             -1 = duplicated leftover node (start of trace)
%                             -2 = duplicated fix-up node (at end of trace)
%       TR,     % 5     % Number of the trace this op is scheduled on
%                             (unbound before op is picked for a trace)
%       PRI,    % 6     % Priority value of this node
%       SCH,    % 7     % Bound to "yes" immediately after this operation is
%                             scheduled; unbound before then
%       PSCH,   % 8     % Bound to "yes" for the instruction after this
%                             operation is scheduled; unbound before then
%       CHLD,   % 9     % List of all children (other operations that have
%                             this one as a pre- or co-requisite)
%       COR,    % 10    % List of nodes that are co-requisite to this one
%       PRE,    % 11    % List of nodes that are pre-requisite to this one
%       FOR,    % 12    % List of nodes that must be scheduled a certain
%                             time after this one, with those delay times
%       USR,    % 13    % user([Cor],[Pre],[For]) where
%                             Cor and Pre list elements = (Block,Index)
%                             For list elements = (Block,Index,Delay)
%                             note: all elements are dependent *children*
%       ULAB,   % 14    % List of user-dependency labels for this node.
%      )
%
%
% structure of a compacted instruction node:
%  instr(operations,    % list of operations within this instruction
%        prereq,        % prerequisite list for forced operations
%        coreq,         % corequisite list for forced operations
%        forced,        % forced list for forced operations
%
%
% temporary structures:
%  block_list           % list of basic blocks along the current trace
%                         of the form blk([Num|Type], Count)
%
%  this_later           % list of operations schedulable now or later
%  next_later           % list of operations schedulable after now
%
%  comp_list            % list of compacted instructions
%
%  fix_up               % tree of nodes of the form
%                         <block # / fix(safe,fixlist,fixgoto,unsafelab)>
%                         where fixlist and fixgoto are as described below;
%                         safe is bound to "safe" after (and if) a safe
%                         block entry point is found;
%                         unsafelab is the unsafe block entry label -- it
%                         is usually bound to "unsafeXXX", but if the unsafe
%                         entry point consists of only an unconditional goto
%                         then unsafelab is bound to that goto's destination
%                         instead (bypasses an extra unnecessary jump).
%
% gross stuff:
%  fix lists     }       One per block; contain lists of instructions of
%                }       that block that were scheduled before the "safe"
%                }       entry point for that block.
%  goto-fix op  }       One per block; contains a goto instruction that
%               }       must be fixed up.  Before such a goto is found,
%               }       this is unbound.  After it is known that there
%               }       is no such goto, this is changed to be a goto
%               }       to the safe entry point for the block.  Thus,
%               }       any time this block is fixed up, this is the
%               }       last instruction that must be scheduled.
%  leftover lists       One per goto instruction; contain lists of
%                       instructions that preceded the goto instruction
%                       but have not yet been scheduled--these must be
%                       inserted at the beginning of all of the secondary
%                       branch destinations.

:- op(  950, xfx, <--). % destructive assignment
:- op(  700, xfx, \=).  % test for value inequality
:- op(  700, xfx, ?).   % test for type equality
:- op(  700, xfx, \?).  % test for type inequality
:- op(  690, fy, ^).    % "pointer to"
:- op(  400, fy, ~).    % bitwise complement operator
:- op(  300, fx, ^^).   % "unbound variable"
:- op(  250, xfx, @).   % concatenation of type and value fields


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% Overall top-level compaction predicates %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Top-level call to compact a program.
%   arg 1: input:  filename of program to compact (without .ucode extension)
%
compact(InName) :-
        make_in_name(InName, NewInName),
        see(NewInName),
        read(P),
        seen,
        write('Preprocessing microoperation list...'), ttyflush,
        unroll_loops(P, U, [], []),
        xref_labels(U, 1, GLabels),
        expand_program(U, E, 1, 1, GLabels, Branches),
        BrList = [br([], 1, start) | Branches],
        nl, write('Compacting traces...'), ttyflush,
        trace_schedule(E, BrList, 1, CompList, _), !,
        tie_up_loose_end(CompList),
        nl, write('Creating compacted instruction list...'), ttyflush,
        extract_instrs(CompList, InstrList),
        name(InName, InList),
        convert_name(InList, OutList),
        name(OutName, OutList),
        tell(OutName),
        write_usage_info,
        write('.'), nl,
        writelst(InstrList),
        write('.'), nl,
        told.
```

```
% Convert a base filename into the filename of the input microcode.
%   arg 1: input:  the base filename (with no extension)
%   arg 2: output: the filename with ".ucode" extension
%
make_in_name(InName, NewInName) :-
        name(InName, InList),
        name('.ucode', InSuffix),
        append(InList, InSuffix, NewInList),
        name(NewInName, NewInList).

% Convert an input filename into an output filename.
%   arg 1: input:  the input filename (with or without ".ucode" extension)
%   arg 2: output: the output filename (with ".compact" extension)
%
convert_name(InSuffix, OutSuffix) :-
        name('.ucode', InSuffix), !,
        name('.compact', OutSuffix).
convert_name([], OutSuffix) :- !,
        name('.compact', OutSuffix).
convert_name([Char | InRest], [Char | OutRest]) :-
        convert_name(InRest, OutRest).


% Run a test program through the compactor and show the results.
%
test :-
        get_program(P),
 write('Original program: '), nl, writelst(P), nl,
        unroll_loops(P, U, [], []),
        xref_labels(U, 1, GLabels),
        expand_program(U, E, 1, 1, GLabels, Branches),
        BrList = [br([], 1, start) | Branches],
        trace_schedule(E, BrList, 1, CompList, _),
        tie_up_loose_end(CompList),
        extract_instrs(CompList, InstrList),
        nl, write('Instruction List: '), nl, writelst(InstrList), nl.

% Some simple test programs for "test" to use.
%
%get_program([  a<--g, b<--e, label(point1), b<--c, g<--d,
%               loadlabel(store, point1), label(point2), c<--e, b<--a,
%               label(point3), d<--a, a<--c,
%               loadlabel(store, point4), label(point4), b<--g, e<--c,
%               goto([[store]]), end]).

get_program([  a<--g, b<--d, c<--b, e<--d, f<--g,
               label(start), a<--b, c<--e, d<--g, g<--b,
               goto([(a,start),fall]), label(fall),
               d<--f, e<--a, b<--c, f<--g, end]).

%get_program([  b<--a, d<--c, goto([(c,dest),fall]),
%               label(fall), e<--a, f<--d,
%               label(dest), f<--a, a<--g, b<--f, end]).

%get_program([  b<--a, d<--c, e<--a, f<--d, f<--a, a<--g, b<--f, end]).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% Pre-compaction program transformations and expansions %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Unroll the innermost loops of a program.
%   arg 1: input:  original list of microoperations
%   arg 2: output: new list of microoperations, with loop code duplicated
%   arg 3: input:  []
%                (the most recent label found; we have found an innermost
%                 loop iff the destination of a goto matches this label)
%   arg 4: input:  []
%                (reverse-list of instructions that must be duplicated if
%                 an inner loop is found and unrolled)
%
unroll_loops([], [], _LoopLabel, _Copy) :- !.
unroll_loops([label(Label)|Rest], [label(Label)|NewRest], _LoopLabel, _Copy) :- !,
        unroll_loops(Rest, NewRest, Label, []).
unroll_loops([goto(DestList)|Rest], RestOut, LoopLabel, Copy) :-
        DestList = [FirstDest|_RestDest],
        extract_label(FirstDest, LoopLabel), !,
        duplicate_loop(1, DestList, LoopLabel, Copy, RestOut, MyRest),
        MyRest = [goto(DestList)|NewRest],
        unroll_loops(Rest, NewRest, [], []).
unroll_loops([goto(DestList)|Rest], [goto(DestList)|NewRest], _LoopLabel, _Copy) :- !,
        unroll_loops(Rest, NewRest, [], []).
unroll_loops([Instr|Rest], [Instr|NewRest], LoopLabel, Copy) :-
        unroll_loops(Rest, NewRest, LoopLabel, [Instr|Copy]).

% Duplicate the code for an innermost loop that is to be unrolled.
%   arg 1: input:  number of times to unroll the loop (# of copies of the code)
%   arg 2: input:  list of the destination labels in the "goto" operation that
%                  ends the inner loop -- since we're duplicating code, we have
%                  to create new labels and update these "goto"s accordingly
%   arg 3: input:  original label of the entry point to the inner loop --
%                  this is used as a base name for constructing new labels
%   arg 4: input:  reverse-list of the instructions to be duplicated
%   arg 5: input:  starting point in open-end list where operations can be added
%   arg 6: output: point in open-end list after all unrolled loop code has
%                  been inserted, where subsequent operations can be added
%
duplicate_loop(0, _DestList, _LoopLabel, _Copy, RestOut, RestOut) :- !.
duplicate_loop(Count, DestList, LoopLabel, Copy, RestOut, NewRest) :-
        reverse(Copy, TempRest, CopyList),
        name(LoopLabel, LabelName),
        name(Count, CountName),
        append(CountName, [42 | LabelName], NewName),
        name(NewLabel, NewName),
        DestList = [FirstDest|RestDest],
        replace_label(FirstDest, NewLabel, NewDest),
        NewDestList = [NewDest|RestDest],
        RestOut = [goto(NewDestList), label(NewLabel) | CopyList],
        NewCount is Count - 1,
        duplicate_loop(NewCount, DestList, LoopLabel, Copy, TempRest, NewRest).
```

```
% Create a cross reference of block labels to block numbers.
%  A new block number is assigned following any goto or end operation.
%   arg 1: input:  list of operations
%   arg 2: input:  first block number (usually 1)
%               (counts the blocks found in the operation list)
%   arg 3: output: tree mapping each original label to its block number
%
xref_labels([], _Block, _Labels) :- !.
xref_labels([label(Label)|Rest], Block, Labels) :- !,
        perm_tree_store(Label, Block, Labels),
        xref_labels(Rest, Block, Labels).
xref_labels([goto(_)|Rest], Block, Labels) :- !,
        NewBlock is Block + 1,
        xref_labels(Rest, NewBlock, Labels).
xref_labels([end|Rest], Block, Labels) :- !,
        NewBlock is Block + 1,
        xref_labels(Rest, NewBlock, Labels).
xref_labels([_Other|Rest], Block, Labels) :-
        xref_labels(Rest, Block, Labels).


% Expand a simple list of operations into a list of operation nodes.
%  This procedure must do the following:
%       Change "goto" operations to use block numbers instead of original names
%       Expand basic operations into full operation nodes
%       Add user-specified constraints to operation nodes
%       Create a branch list of unsafe entry points
%   arg 1: input:  original list of operations
%   arg 2: ouptut: expanded list of operation nodes
%   arg 3: input:  first block number (usually 1)
%               (counts the blocks found in the operation list)
%   arg 4: input:  1
%               (operation's index number within its block)
%   arg 5: input:  tree mapping each original label to its block number
%   arg 6: output: branch list, containing all unsafe block entry points
%
                        % All done when input operation list is empty
expand_program([], [], _Block, _Index, _Labels, BrList) :- !,
        tie_up_loose_end(BrList).
                        % At an end node, start a new block
expand_program([end|Rest], Nodes0, Block, Index, Labels, BrList) :- !,
        Nodes0 = [node(end,Block,Index,1,_,_,_,_,_,_,_,UsrDep,UsrLab) | Nodes1],
        no_more_user_dep(UsrDep, UsrLab),
        NewBlock is Block + 1,
        expand_program(Rest, Nodes1, NewBlock, 1, Labels, BrList).
                        % At a goto node, start new block
expand_program([goto(List)|Rest], Nodes0, Block, Index, Labels, BrList) :- !,
        (List = [[_Indirect]] ->
                (NewList = List
                ) ;
                (number_gotos(List, NewList, Labels, BrList)
                )
        ),
        Nodes0 = [node(goto(NewList),Block,Index,1,
                       _,_,_,_,_,_,_,UsrDep,UsrLab) | Nodes1],
        no_more_user_dep(UsrDep, UsrLab),
        NewBlock is Block + 1,
        expand_program(Rest, Nodes1, NewBlock, 1, Labels, BrList).
                        % In a loadlabel node, update the destination label
expand_program([loadlabel(Name,Lab)|Rest], Nodes0, Block, Index, Labels, BrList) :- !,
        number_gotos([Lab], [NewLab], Labels, BrList),
        unsafe_add_label(NewLab, UnsafeLab),
        Nodes0 = [node(loadlabel(Name,UnsafeLab),Block,Index,1,
                  _,_,_,_,_,_,_,UsrDep,UsrLab) | Nodes1],
        no_more_user_dep(UsrDep, UsrLab),
        NewIndex is Index + 1,
        expand_program(Rest, Nodes1, Block, NewIndex, Labels, BrList).
                        % At a label node, do a sanity check
expand_program([label(Label)|Rest], Nodes0, Block, Index, Labels, BrList) :- !,
        perm_tree_lookup(Label, LabBlock, Labels),
        (LabBlock == Block ->
                (true
                ) ;
                (write('*** ERROR in expand_program: label '), write(Label),
                 write(': block '), write(Block),
                 write(' doesn''t match block '), write(LabBlock), nl,
                 fail
                )
        ),
        expand_program(Rest, Nodes0, Block, Index, Labels, BrList).
                        % Add dependencies to next operation node
expand_program([{CoReq,PreReq,Forced}|Rest], Nodes0, Block, Index, Labels, BrList) :- !,
        Nodes0 = [node(_,Block,Index,_,_,_,_,_,_,_,_,
                       user(CoList,PreList,ForList),_) | _Nodes1],
        add_to_open_end(CoReq, CoList),
        add_to_open_end(PreReq, PreList),
        add_to_open_end(Forced, ForList),
        expand_program(Rest, Nodes0, Block, Index, Labels, BrList).
expand_program([{Marker}|Rest], Nodes0, Block, Index, Labels, BrList) :- !,
        Nodes0 = [node(_,Block,Index,_,_,_,_,_,_,_,_,_,DepLabels) | _Nodes1],
        add_to_open_end(Marker, DepLabels),
        expand_program(Rest, Nodes0, Block, Index, Labels, BrList).
                        % For a regular op, create node & increment index
expand_program([Op|Rest], Nodes0, Block, Index, Labels, BrList) :-
        Nodes0 = [node(Op,Block,Index,1,_,_,_,_,_,_,_,UsrDep,UsrLab) | Nodes1],
        no_more_user_dep(UsrDep, UsrLab),
        NewIndex is Index + 1,
        expand_program(Rest, Nodes1, Block, NewIndex, Labels, BrList).

% Translate destination labels into block numbers in a goto destination list.
%   arg 1: input:  original list of (possibly conditional) destinations
%   arg 2: output: new list, with same conditions, but block-numbered labels
%   arg 3: input:  tree of mappings from symbolic label names to block numbers
%
number_gotos([], [], _Labels, _BrList) :- !.
number_gotos([First|Rest], [NFirst|NRest], Labels, BrList) :-
        extract_label(First, Dest),
        perm_tree_lookup(Dest, Num, Labels),
        replace_label(First, Num, NFirst),
        unsafe_add_label(Num, UnsafeNum),
        add_to_open_end(br([], Num, UnsafeNum), BrList),
        number_gotos(Rest, NRest, Labels, BrList).

% Finish off all user-specified dependency information in a completed node.
%   arg 1: input:  user dependency information field from completed node
%   arg 2: input:  user-given label list for this completed node
%
no_more_user_dep(user(UCor,UPre,UFor), DepLabels) :-
        tie_up_loose_end(UCor),
        tie_up_loose_end(UPre),
        tie_up_loose_end(UFor),
        tie_up_loose_end(DepLabels).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%% Top-level trace scheduling code %%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Perform trace scheduling on a program; compact all possible code sequences.
%   arg 1: input:  complete original program of operation nodes
%   arg 2: input:  list of branch points -- we have yet to compact the code
%                  starting at each of these branch points
%   arg 3: input:  first trace number (usually 1)
%               (holds the number of the trace currently being worked on)
%   arg 4: output: list of the compacted instruction nodes
%   arg 5: input:  tree mapping block numbers to lists of operations that
%                  must be "fixed up" anytime a trace wants to jump there
%
trace_schedule(_Prog, [], _TrNum, _CompList, _FixUp) :- !.
trace_schedule(Prog, [Br|BrL0], TrNum, CompL0, Fix0) :-
    print(TrNum), print('..'), ttyflush,
        Br = br(Trace, Block, Label),
        duplicate_op_list(Trace, DupTrace),
        find_block_start(Prog, Block, Nodes),
        append(DupTrace, Nodes, TotTrace),
        start_block_list(BlIn),
        pick_trace(Prog, TotTrace, TrNum, [], [Terminal|BackTrace], BlIn, BlOut),
        end_block_list(BlOut, Blocks),
        (Terminal = [] ->
                reverse(BackTrace, [], ForwTrace) ;
                (reverse([Terminal|BackTrace], [], ForwTrace),
                 arg(6, Terminal, 0)
                )
        ),
        add_dependencies(ForwTrace, _, _, [], []),
%       arg(6, Terminal, 0),
        assign_priorities(BackTrace, [], Ready, [], Later),
        LabelNode = node(label(Label),0,0,0,TrNum,_,_,_,_,_,_,_,_,_),
        schedule_op_instr(LabelNode, CompL0),
        first_schedule_trace(Ready, Later, Terminal, CompL0, CompList,
                BrL0, BrList, [blk([0|0],0)|Blocks], ForwTrace, Fix0, FixUp), !,
        NewTrNum is TrNum + 1,
        trace_schedule(Prog, BrList, NewTrNum, CompList, FixUp).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%% Pick out a most-likely trace to be scheduled %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Follow through a program, finding an unscheduled trace.
%  The trace ends with:
%   The "end" node, if an "end" node is encountered.
%   An indirect "goto" node, if one is encountered.
%   A "goto safe_" node, if primary destination has been compacted; in this
%      case, "goto trace_from_to_" and "fixup_" nodes are also used.
%   -- gotos to uncompacted nodes simply continue the trace (with a
%      "goto trace_from_to_" instruction).
%   -- all secondary goto destinations are relabeled in the
%      "trace_from_to_" format, but will be added to the branch list in
%      "schedule_trace" (when the set of leftover nodes is known).
%   -- "schedule_trace" must also generate "label" nodes to match
%      each "goto trace_from_to_" node, as well as "safe_" labels.
%   arg 1: input:  the complete list of program operation nodes
%   arg 2: input:  the list of nodes yet to be considered on this trace
%   arg 3: input:  the current trace number
%   arg 4: input:  []
%               (reverse list of operation nodes chosen for this trace so far)
%   arg 5: output: reverse list of all operation nodes in the trace
%   arg 6: input:  starting count of nodes within each block number
%   arg 7: output: final count of nodes within each block number
%
pick_trace(_Prog, Nodes, TrNum, Tr0, Trace, BlList, BlList) :-
        first_op(Nodes, Node),
        arg(5, Node, NodeTrace),
                        % End trace, and fix up, upon hitting compacted node
        nonvar(NodeTrace), !,
        arg(2, Node, DestBlock),
        FixNode = node(fixup(DestBlock),0,0,0,TrNum,_,_,_,_,_,_,_,user([],[],[]),[]),
        Trace = [[], FixNode | Tr0].
pick_trace(_Prog, Nodes, TrNum, Tr0, Trace, BlL0, BlList) :-
        first_op(Nodes, Node),
                        % End trace at an "end" or "indirect goto" node
        (arg(1, Node, end) ; arg(1, Node, goto([[_]]))), !,
        arg(5, Node, TrNum),            % set trace number
        arg(2, Node, ThisBlock),        % find block # of this op
        arg(4, Node, ThisType),
        block_list(BlL0, [ThisBlock|ThisType], BlList),
        Trace = [Node | Tr0].
pick_trace(Prog, Nodes, TrNum, Tr0, Trace, BlL0, BlList) :-
        first_op(Nodes, Node),
        arg(1, Node, goto(List)), !,    % branch in program flow
        arg(5, Node, TrNum),            % set trace number
                        % Relabel all destinations in "from_to_" format
        Node = node(_OP,BLK,IDX,TYP,_TR,PRI,SCH,PSCH,CHLD,COR,PRE,FOR,USR,ULAB),
        name(BLK, BlkName),
        append("from", BlkName, FromBlkName),
        replace_labels(List, FromBlkName, NewList),
        NewNode = node(goto(NewList),BLK,IDX,TYP,TrNum,
                        PRI,SCH,PSCH,CHLD,COR,PRE,FOR,USR,ULAB),
        block_list(BlL0, [BLK|TYP], BlL1),
                        % Use primary label to find next nodes in the trace
        List = [PrimDest | _OtherDest],
        extract_label(PrimDest, PrimLabel),
        find_block_start(Prog, PrimLabel, DestNodes),
        pick_trace(Prog, DestNodes, TrNum, [NewNode | Tr0], Trace, BlL1, BlList).
pick_trace(Prog, Nodes, TrNum, Tr0, Trace, BlL0, BlList) :-
                        % An ordinary operation node, just move on to next one
        first_op(Nodes, Node),
        arg(5, Node, TrNum),           % set trace number
        arg(2, Node, ThisBlock),       % find block # of this op
        arg(4, Node, ThisType),
        block_list(BlL0, [ThisBlock|ThisType], BlL1),
        rest_op(Nodes, NewNodes),
        pick_trace(Prog, NewNodes, TrNum, [Node | Tr0], Trace, BlL1, BlList).


% Find the point in the operation node list where a block starts.
%   arg 1: input:  the complete list of operation nodes
%   arg 2: input:  the block number being searched for
%   arg 3: output: list of operation nodes starting at beginning of the block
%
find_block_start([Node|Rest], Blkno, Nodes) :-
        arg(2, Node, Blkno), !,
        Nodes = [Node|Rest].
find_block_start([_Node|Rest], Blkno, Nodes) :-
        find_block_start(Rest, Blkno, Nodes).

% Return the first operation of an operation list
%   arg 1: input:  the operation list
%   arg 2: output: the first operation of that list
%
first_op([Node|_Rest], Node).

% Return all operations after the first from an operation list
%   arg 1: input:  the operation list
%   arg 2: output: the non-first operations of that list
%
rest_op([_Node|Rest], Rest).


%%%%%%%%%%%%%% Routines for manipulating goto destination labels %%%%%%%%%%%%

% Relabel the destinations of a goto list to include the block number.
%   arg 1: input:  original list of (possibly conditional) goto destinations
%   arg 2: input:  string containing source block number (usually "fromXXX")
%   arg 3: output: new list of relabeled goto destinations
%
replace_labels([], _SourceName, []) :- !.
replace_labels([First|Rest], SourceName, [RFirst|RRest]) :-
        extract_label(First, ToNum),
        name(ToNum, ToName),
        append("to", ToName, DestName),
        append(SourceName, DestName, NewName),
        name(NewLabel, NewName),
        replace_label(First, NewLabel, RFirst),
        replace_labels(Rest, SourceName, RRest).
```

```
% Strip the "fromXXXto" part off of a label, giving the dest block #.
%   arg 1: input:  the original "fromXXXtoYYY" label
%   arg 2: output: the "YYY" part of the label
%
unreplace_label(Old, New) :-
        name(Old, OldName),
        append(_From, [116, 111 | NewName], OldName),
        name(New, NewName).

% Extract destination numbers YYY from list of "fromXXXtoYYY" labels, and
%  look up the safe/unsafe labels for these dests from the fix up structure.
%   arg 1: input:  list of the original "fromXXXtoYYY" labels
%   arg 2: input:  the current fix-up list
%   arg 3: output: list of new labels, later bound to "safeYYY" or "unsafeYYY"
%
unsafe_replace_labels([], _, []) :- !.
                % Indirect goto destinations are not affected.
unsafe_replace_labels([[Indirect]], _, [[Indirect]]) :- !.
unsafe_replace_labels([Old|OldRest], FixUp, [New|NewRest]) :-
        extract_label(Old, OldLabel),
        name(OldLabel, OldName),
        append(_From, [116, 111 | DestName], OldName),
        name(DestNum, DestName),
        perm_tree_store(DestNum, fix(_Safe,_FixList,_FixGoto,UnsafeLab), FixUp),
        replace_label(Old, UnsafeLab, New),
        unsafe_replace_labels(OldRest, FixUp, NewRest).

% Add "unsafe" to a goto destination block number.
%   arg 1: input:  the original destination block number "YYY"
%   arg 2: output: the new "unsafeYYY" label
%
unsafe_add_label(OldLabel, NewLabel) :-
        name(OldLabel, OldName),
        append("unsafe", OldName, NewName),
        name(NewLabel, NewName).


%%%%%%%%%%%%%%%%%%%% Routines to manipulate "block lists" %%%%%%%%%%%%%%%%%%%%
%%%% (the list of block numbers and node counts along the current trace) %%%%

% Create an empty block list count structure.
%   arg 1: output: the empty block list structure
start_block_list(unblist).

% Copy a block list, removing any zero-count blocks.
%   arg 1: input:  the original block list
%   arg 2: input:  an empty block list
%               (holds intermediate lists as they are being built)
%   arg 3: output: the duplicated block list
%
restart_block_list([], BList, BList) :- !.
restart_block_list([blk(_,0)|Rest], BL0, BList) :- !,
        restart_block_list(Rest, BL0, BList).
restart_block_list([blk(BlockType,Count)|Rest], Bl0, BList) :-
        block_list(Bl0, BlockType, Bl1),
        NewCount is Count - 1,
        restart_block_list([blk(BlockType,NewCount)|Rest], Bl1, BList).

% Account for a new node in the block list count structure.
%    arg 1: input:  the original block list count structure
%    arg 2: input:  the block number/type of the trace's new operation
%    arg 3: output: the new block list count structure
%
block_list(unblist, Num, blist(Blk-Blk, Num, 1)) :- !.
block_list(blist(BList, Num, Count), Num, blist(BList, Num, NewCount)) :- !,
        NewCount is Count + 1.
block_list(blist(BHead-[blk(Num, Count)|BTail], Num, Count),
                NewNum, blist(BHead-BTail, NewNum, 1)).

% Account for a list of new nodes, in the block list count structure.
%    arg 1: input:  the original block list count structure
%    arg 2: input:  list of block numbers/types of the trace's new operation
%    arg 3: output: the new block list count structure
%
list_block_list(BList, [], BList) :- !.
list_block_list(BListIn, [First | Rest], BListOut) :-
        arg(2, First, Block),
        arg(4, First, Type),
        block_list(BListIn, [Block|Type], BListTemp),
        list_block_list(BListTemp, Rest, BListOut).

% Finish and clean up a block list count structure after it has been built.
%    arg 1: input:  the block list count structure after counting all blocks
%    arg 2: output: the cleaned-up, pure list of blocks/types along trace
%
end_block_list(blist(BHead-[blk(Num, Count)], Num, Count), BHead) :- !.
end_block_list(unblist, []).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% Assign scheduling priorities to each node in the trace %%%%%%%%%
%%%%% (also produce list of parent-less (ready to execute) operations) %%%%%

% Assign priorities to all operation nodes in a list.
%    arg 1: input:  reverse list of operation nodes
%    arg 2: input:  list of nodes already known to be ready now (usually [])
%    arg 3: output: final list of all nodes that are ready now
%    arg 4: input:  list of nodes already knwon to be ready later (usually [])
%    arg 5: output: final list of all nodes that are ready later
%
assign_priorities([], Ready, Ready, Later, Later) :- !.
assign_priorities([Head|Rest], R0, Ready, L0, Later) :-
        assign_priority(Head, R0, R1, L0, L1),
        assign_priorities(Rest, R1, Ready, L1, Later).

% Assign priority to one operation node.
%    arg 1: input:  operation node that is to receive a priority value
%    arg 2: input:  initial list of nodes known to be ready now
%    arg 3: output: final list of nodes that are ready now (may include this one)
%    arg 4: input:  initial list of nodes knwon to be ready later
%    arg 5: output: final list of nodes that are ready later (may include this)
```

```
%
assign_priority(Node, R0, Ready, L0, Later) :-
        arg(9, Node, Children),
        max_next_priority(Children, TNLPri),
        arg(12, Node, Forced),
        max_forced_priority(Forced, FPri),
        max(TNLPri, FPri, Pri),
        arg(6, Node, Pri),              % Assign node's priority
        find_ready_child([Node], R1, L1),
        priority_merge_list(R0, R1, Ready),
        priority_merge_list(L0, L1, Later).


%%%%%%%%%% Routines for working with the priority fields of nodes %%%%%%%%%%

% Calculate a node's priority from the priorities of its regular children.
%   arg 1: input:  (possibly open-ended) list of the node's children
%   arg 2: output: node's priority justified by its children
%               (one more than the highest child priority value found)
%
max_next_priority(Var, 0) :-
        var(Var), !.
max_next_priority([], 0) :- !.
max_next_priority([First | Rest], Max) :-
        arg(6, First, FPri),
        FPri1 is FPri + 1,
        max_next_priority(Rest, RPri),
        max(FPri1, RPri, Max).

% Calculate a node's priority from the priorities of its forced children.
%   arg 1: input:  (possibly open-ended) list of the node's forced operations
%   arg 2: output: node's priority justified by its forced operations
%               (the largest sum of a forced node's priority and its delay)
%
max_forced_priority(Var, 0) :-
        var(Var), !.
max_forced_priority([], 0) :- !.
max_forced_priority([force(Delay,First) | Rest], Max) :-
        arg(6, First, FPri),
        max_forced_priority(Rest, RPri),
        ThisPri is Delay+FPri,
        max(ThisPri, RPri, Max).


%%%%%%%%%% Insert node into the proper position in a priority queue %%%%%%%%%

% Insert one node into its proper position in a priority queue.
%   arg 1: input:  the node to be inserted
%   arg 2: input:  the original list of nodes (sorted in decreasing priority)
%   arg 3: output: the new list of nodes, with arg 1 in its proper place
%
priority_merge(Node, [], [Node]) :- !.
priority_merge(Node, [First | Rest], Ready) :-
        arg(6, Node, NPri),
        arg(6, First, FPri),
        NPri > FPri, !,
        Ready = [Node, First | Rest].
priority_merge(Node, [First | Rest], [First | Ready]) :-
        priority_merge(Node, Rest, Ready).

% Insert a list of nodes into their proper positions in a priority queue.
%   arg 1: input:  list of nodes to be inserted
%   arg 2: input:  the original list of nodes (sorted in decreasing priority)
%   arg 3: output: the new list of nodes, with new nodes in their proper places
%
priority_merge_list([], List, List) :- !.
priority_merge_list([First|Rest], L0, List) :-
        priority_merge(First, L0, L1),
        priority_merge_list(Rest, L1, List).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% Add data dependencies to the nodes in the trace %%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Data dependency determination:
%  reg read 1 -- reg read 2     -> no restriction
%  reg read 1 -- reg write 2    -> 2 must occur no earlier than 1
%  reg write 1 -- reg read 2    -> 2 must occur later than 1
%  reg write 1 -- reg write 2   -> 2 must occur later than 1
% memory:
%  When referenced by an address that is an offset from a register
%  (displacement mode), each distinct <base register,offset> combination
%  is treated as a "register" using the rules above.  For example,
%  "h<--mem(s), mem(s)<--t" would have a read-write dependency; but
%  "mem(s)<--h, mem(s+1)<--t" would have *no* memory dependencies.
%  When a base register is changed (for example, "s<--s+2") the following
%  things happen:
%    * All previous uses of the register as a base register (which are
%      considered as "reads" of the register) receive read-write dependencies
%      with the instruction that modifies the register.
%    * All subsequent uses of the register as a base register receive
%      write-read dependencies on the instruction that modified the register.
%    * Because the above two rules automatically serialize all memory
%      accesses using the base register *after* the modification with the
%      accesses *before* the modification, we can just clear out the read &
%      write access lists for *all* offsets from the base register.
% Memory references to absolute locations work similarly -- each different
% location is treated as a distinct "register".  Of course, in this case,
% there is no "base register" that can be changed.
% This policy implicitly assumes that different base registers define
% unique, non-overlapping segments of memory; this is generally a valid
% assumption in Prolog.
% But this policy has difficulty with "aliases", or registers that "point"
% into memory.  The compactor will not detect dependencies between direct
% accesses to the memory and accesses to the register.  The only automatic
% solution (that I can see) to this problem would also place many
% unnecessary constraints between operations that will really never
% interfere with each other.  And, since this situation seems to cause
% problems relatively infrequently, the compactor requires that such
% dependencies be specified *manually* in the input file.
```

```
%

% Implementation:
%  "Reads" and "Writes" are (temporary) trees that map register or memory
%  names into lists of nodes that last read (or wrote) them.
%  Register names map directly into an accessor list.
%  Memory names of the form "mem(<BaseRegister>)" (or "mem([])" for absolute
%  accesses) map into another tree; in this second tree, the offset number
%  maps into the accessor list.  This method allows entire base-referenced
%  sections of memory to be easily "wiped clean", as described above.

% Add data-, goto-, set/show-, and user-specified dependencies to trace nodes.
%   arg 1: input:  list of all nodes in the trace
%   arg 2: input:  _
%             (temp. tree of registers/memory, to nodes that last read them)
%   arg 3: input:  _
%             (temp. tree of registers/memory, to nodes that last wrote them)
%   arg 4: input:  []
%             (most recent "goto" node found along this trace, [] for none)
%   arg 5: input:  []
%             (most recent "set" or "show" node along trace, [] for none)
%
add_dependencies([], _Reads, _Writes, _GotoNode, _SetNode) :- !.
add_dependencies([Node | Rest], R0, W0, GotoN0, SetN0) :-
        arg(1, Node, Op),
        data_use(Op, ORead, OWrite),
        read_depend(ORead, Node, R0, W0, R1),
        write_depend(OWrite, Node, R1, W0, R2, W1),
        clear_mem_depend(Op, R2, Reads, W1, Writes),
        goto_depend(Op, Node, GotoN0, GotoNode),
        set_depend(Op, Node, SetN0, SetNode),
        user_depend(Node, Rest),
        add_dependencies(Rest, Reads, Writes, GotoNode, SetNode).


%%%%%%%%%%%%%%% Routines for determining register/memory usage %%%%%%%%%%%%%%%

% Determine the registers/memory locations read or written by an operation.
%   arg 1: input:  the operation to be examined
%   arg 2: output: the list of registers/memory locations read by operation
%             (memory locations are returned in mem(basereg,offset) form)
%   arg 3: output: the list of registers/memory locations written by operation
%             (memory locations are returned in mem(basereg,offset) form)
%
data_use(Dest<--Src, Reads, Writes) :- !,
        src_use(Src, [], R1),
        dest_use(Dest, R1, Reads, [], Writes).
data_use(goto(GotoList), Reads, []) :- !,
        goto_use(GotoList, [], Reads).
data_use(set(SetList), [], Writes) :- !,
        set_use(SetList, [], Writes).
data_use(show(ShowList), Reads, []) :- !,
        show_use(ShowList, [], Reads).
data_use(loadlabel(Dest, _Name), Reads, Writes) :- !,
        dest_use(Dest, [], Reads, [], Writes).
data_use(label(_), [], []) :- !.
data_use(fixup(_), [], []) :- !.
data_use(end, [], []) :- !.
data_use(Unknown, [], []) :-
        write('*** WARNING: unknown op '), write(Unknown), nl.

% Determine the registers/memory locations read by the source of a transfer.
%   arg 1: input:  "source" expression from a transfer operation
%   arg 2: input:  list of registers/memory locations already read by operation
%   arg 3: output: list of read locations, including this "source"
%
src_use(_Tag@Term, R0, Reads) :- !,
        src_use(Term, R0, Reads).
src_use(Atom, R0, [Atom | R0]) :-
        atom(Atom), !.
src_use(mem(Addr), R0, Reads) :- !,
        addr_use(Addr, [], BaseReg, +, 0, Offset),
        Reads = [BaseReg, mem(BaseReg,Offset) | R0].
src_use(Op, R0, Reads) :-
        functor(Op, _, 1), !,
        arg(1, Op, Term),
        src_use(Term, R0, Reads).
src_use(Op, R0, Reads) :-
        functor(Op, _, 2), !,
        arg(1, Op, Term1),
        arg(2, Op, Term2),
        src_use(Term1, R0, R1),
        src_use(Term2, R1, Reads).
src_use(Number, Reads, Reads) :-
        number(Number), !.
src_use(Unknown, Reads, Reads) :-
        write('*** ERROR: invalid source '), write(Unknown), nl.

% Determine the registers/memory locations read & written by a destination.
%   arg 1: input:  "destination" expression from a transfer operation
%   arg 2: input:  list of registers/memory locations already read by operation
%   arg 3: output: list of read locations, including this "destination"
%   arg 4: input:  list of registers/memory already written by operation
%   arg 5: output: list of written locations, including this "destination"
%
dest_use(Term, Reads, Reads, Writes, [Term|Writes]) :-  % Store into register
        atom(Term), !.
                                                   % Store into memory location
dest_use(mem(Addr), Reads, [BaseReg|Reads], Writes, [mem(BaseReg,Offset)|Writes]) :- !,
        addr_use(Addr, [], BaseReg, +, 0, Offset).
dest_use(Unknown, Reads, Reads, Writes, Writes) :-
        write('*** ERROR: invalid destination '), write(Unknown), nl.

% Determine the registers/memory locations read in doing a memory reference;
%  like "src_use", but restricted to displacement-type arithmetic (+ and -).
%   arg 1: input:  "address" expression from a memory reference
%   arg 2: input:  []
%              (name of last found base register in "address", or [] if none)
%   arg 3: output: name of last base register in "address"; [] if none
%   arg 4: input:  +
%              (+ or -, whether next addr. term should be added or subtracted)
%   arg 5: input:  0
%               (current offset value in "address")
%   arg 6: output: total offset value in "address"
%
addr_use(_Tag@Term, BaseReg0, BaseReg, Sign, Offset0, Offset) :- !,
        addr_use(Term, BaseReg0, BaseReg, Sign, Offset0, Offset).
addr_use(Number, BaseReg, BaseReg, Sign, Offset0, Offset) :-
        number(Number), !,
        (Sign == + ->
                Offset is Offset0 + Number ;
                Offset is Offset0 - Number
        ).
addr_use(Atom, BaseReg0, Atom, Sign, Offset, Offset) :-
        atom(Atom), !,
        (BaseReg0 == [] ->
                true ;
                (write('*** WARNING: memory address specified >1 base register; '),
                 write(BaseReg0), write(' is being ignored'), nl
                )
        ),
        (Sign == + ->
                true ;
                (write('*** WARNING: subtracting base register '),
                 write(Atom), write('; I''m adding it instead.'), nl
                )
        ).
addr_use(Op, BaseReg0, BaseReg, Sign, Offset0, Offset) :-
        functor(Op, +, 2), !,
        arg(1, Op, Term1),
        arg(2, Op, Term2),
        addr_use(Term1, BaseReg0, BaseReg1, Sign, Offset0, Offset1),
        addr_use(Term2, BaseReg1, BaseReg, Sign, Offset1, Offset).
addr_use(Op, BaseReg0, BaseReg, Sign, Offset0, Offset) :-
        functor(Op, -, 2), !,
        arg(1, Op, Term1),
        arg(2, Op, Term2),
        addr_use(Term1, BaseReg0, BaseReg1, Sign, Offset0, Offset1),
        (Sign == + ->
                OppSign = - ;
                OppSign = +
        ),
        addr_use(Term2, BaseReg1, BaseReg, OppSign, Offset1, Offset).
addr_use(Unknown, BaseReg, BaseReg, _Sign, Offset, Offset) :-
        write('*** WARNING: unknown memory address form '),
        write(Unknown), write('; I''m ignoring it'), nl.


% Determine the registers/memory locations read by a "goto" operation.
%   arg 1: input:  list of (possibly conditional) goto destinations
%   arg 2: input:  list of registers/memory locations already read by operation
%   arg 3: output: list of read locations, including this goto list
%
goto_use([], Reads, Reads) :- !.
goto_use([Var], Reads, Reads) :-
        var(Var), !.            % Destination not yet finalized
goto_use([[Indirect]], R0, Reads) :- !,
        src_use(Indirect, R0, Reads).
```

```
goto_use([(Cond,_Dest)|Rest], R0, Reads) :- !,
        src_use(Cond, R0, R1),
        goto_use(Rest, R1, Reads).
goto_use([_Dest|Rest], R0, Reads) :- !,
        goto_use(Rest, R0, Reads).

% Determine the registers/memory locations written by a "set" operation.
%   arg 1: input:  list of set assignments
%   arg 2: input:  list of registers/memory locations already written
%   arg 3: output: list of written locations, including this set list
%
set_use([], Writes, Writes) :- !.
set_use([(Dest=_Source) | Rest], W0, Writes) :-
        dest_use(Dest, [], _Reads, W0, W1),
        set_use(Rest, W1, Writes).

% Determine the registers/memory locations read by a "show" operation.
%   arg 1: input:  list of show locations
%   arg 2: input:  list of registers/memory locations already read
%   arg 3: output: list of read locations, including this set list
%
show_use([], Reads, Reads) :- !.
show_use([{_Atom} | Rest], R0, Reads) :-
        show_use(Rest, R0, Reads).
show_use([Source | Rest], R0, Reads) :-
        src_use(Source, R0, R1),
        show_use(Rest, R1, Reads).


%%%% Routines for determining read and write dependencies of an operation %%%%

% From an operation's list of reads, determine its read dependencies.
%   arg 1: input:  list of registers/memory locations read by the operation
%   arg 2: input:  pointer to operation node, used for adding dependencies
%   arg 3: input:  original tree of values and the last nodes to read them
%   arg 4: input:  tree of values and the last nodes to write them
%   arg 5: output: new tree of values and the last nodes that read them
%
read_depend([], _Node, Reads, _Writes, Reads) :- !.
read_depend([First | Rest], Node, R0, Writes, Reads) :-
        temp_tree_var_lookup(First, WList, Writes),
        add_to_all_prereq(Node, WList),
        temp_tree_var_lookup(First, RList, R0),
        temp_tree_var_store(First, [Node | RList], R0, R1),
        read_depend(Rest, Node, R1, Writes, Reads).

% From an operation's list of writes, determine its write dependencies.
%   arg 1: input:  list of registers/memory locations written by the operation
%   arg 2: input:  pointer to operation node, used for adding dependencies
%   arg 3: input:  original tree of values and the last nodes to read them
%   arg 4: input:  original tree of values and the last nodes to write them
%   arg 5: output: new tree of values and the last nodes that read them
%   arg 6: output: new tree of values and the last nodes to write them
%
write_depend([], _Node, Reads, Writes, Reads, Writes) :- !.
write_depend([First | Rest], Node, R0, W0, Reads, Writes) :-
```

```
        temp_tree_var_lookup(First, RList, R0),
        (RList == [] ->
                (temp_tree_var_lookup(First, WList, W0),
                 add_to_all_prereq(Node, WList),
                 temp_tree_var_store(First, [Node], W0, W1),
                 temp_tree_var_store(First, [], R0, R1)
                ) ;
                (temp_tree_var_lookup(First, RList, R0),
                 add_to_all_coreq(Node, RList),
                 temp_tree_var_store(First, [Node], W0, W1),
                 temp_tree_var_store(First, [], R0, R1)
                )
        ),
        write_depend(Rest, Node, R1, W1, Reads, Writes).


% If updating a base register, erase memory dependencies for that base reg.
%   arg 1: input:  the current operation
%   arg 2: input:  original tree of values and the last nodes to read them
%   arg 3: output: new tree of values and the last nodes that read them
%   arg 4: input:  original tree of values and the last nodes to write them
%   arg 5: output: new tree of values and the last nodes to write them
%
clear_mem_depend((Dest <-- _Src), R0, Reads, W0, Writes) :-
        atom(Dest), !,                   % Clear out read & write dependencies
        temp_tree_store(mem(Dest), [], R0, Reads),
        temp_tree_store(mem(Dest), [], W0, Writes).
                                         % Do nothing if not store to register
clear_mem_depend(_Op, Reads, Reads, Writes, Writes).

% If this is a "goto" node, make it the "last goto node found"; in any case,
%  add a dependency from the last goto node(s) to the current node
%   arg 1: input:  the current operation
%   arg 2: input:  pointer to the current node
%   arg 3: input:  the current list of the last goto node(s) found
%   arg 4: output: the new list of the last goto node found
%
goto_depend(goto(_), Node, GotoNode, [Node]) :- !,
                        % This node becomes the "goto node" for subsequent ops
        add_to_all_prereq(Node, GotoNode).
goto_depend(_, Node, GotoNode, GotoNode) :-
        add_to_all_prereq(Node, GotoNode).

% If this is a "set" or "show" node, give it a dependency on the previous
%  "set" or "show" node, and make this node the "last set/show node found"
%   arg 1: input:  the current operation
%   arg 2: input:  pointer to the current node
%   arg 3: input:  the current list of the last set/show node(s) found
%   arg 4: output: the new list of the last set/show node found
%
set_depend(set(_), Node, SetNode, [Node]) :- !,
                        % Each "set" or "show" node depends on the previous one
        add_to_all_coreq(Node, SetNode).
set_depend(show(_), Node, SetNode, [Node]) :- !,
        add_to_all_coreq(Node, SetNode).
set_depend(_, _Node, SetNode, SetNode).
```

```
% Add user-supplied depencency constraints to the current node.
%   arg 1: input:  the current node
%   arg 2: input:  the total trace being scheduled
%
user_depend(Node, Trace) :-
        arg(4, Node, Type),
        arg(13, Node, user(UCor,UPre,UFor)),
        user_co_depend(UCor, Type, Node, Trace),
        user_pre_depend(UPre, Type, Node, Trace),
        user_for_depend(UFor, Type, Node, Trace).

% Add user-supplied corequisite constraints from current node to others.
%   arg 1: input:  list of user-specified corequisites
%   arg 2: input:  "type" of the current node --
%                  there can be no requisites between nodes of different types
%   arg 3: input:  pointer to the current node
%   arg 4: input:  the complete trace currently being examined
%
user_co_depend([], _Type, _Node, _Trace) :- !.
user_co_depend([First|Rest], Type, Node, Trace) :-
        find_user_depend(Trace, Type, First, DepNode), !,
        add_to_all_coreq(DepNode, [Node]),
        user_co_depend(Rest, Type, Node, Trace).
user_co_depend([_First|Rest], Type, Node, Trace) :-
        user_co_depend(Rest, Type, Node, Trace).

% Add user-supplied prerequisite constraints from current node to others.
%   arg 1: input:  list of user-specified prerequisites
%   arg 2: input:  "type" of the current node --
%                  there can be no requisites between nodes of different types
%   arg 3: input:  pointer to the current node
%   arg 4: input:  the complete trace currently being examined
%
user_pre_depend([], _Type, _Node, _Trace) :- !.
user_pre_depend([First|Rest], Type, Node, Trace) :-
        find_user_depend(Trace, Type, First, DepNode), !,
        add_to_all_prereq(DepNode, [Node]),
        user_pre_depend(Rest, Type, Node, Trace).
user_pre_depend([_First|Rest], Type, Node, Trace) :-
        user_pre_depend(Rest, Type, Node, Trace).

% Add user-supplied forced constraints from current node to others.
%   arg 1: input:  list of user-specified forced constraints
%   arg 2: input:  "type" of the current node --
%                  there can be no requisites between nodes of different types
%   arg 3: input:  pointer to the current node
%   arg 4: input:  the complete trace currently being examined
%
user_for_depend([], _Type, _Node, _Trace) :- !.
user_for_depend([(First,Delay)|Rest], Type, Node, Trace) :-
        find_user_depend(Trace, Type, First, DepNode), !,
        arg(12, Node, Forced),
        add_to_open_end((DepNode, Delay), Forced),
        user_for_depend(Rest, Type, Node, Trace).
user_for_depend([_First|Rest], Type, Node, Trace) :-
```

```
        user_for_depend(Rest, Type, Node, Trace).

% Search through the current trace to find a dependent node; fail if none.
%   arg 1: input:  the complete trace being examined
%   arg 2: input:  the type of the node enforcing the dependency
%   arg 3: input:  the user-specified name of the dependent node
%   arg 4: output: the dependent node
%
find_user_depend([Node|_Rest], Type, Name, DepNode) :-
        arg(4, Node, Type),
        arg(14, Node, Name), !,
        DepNode = Node.
find_user_depend([_Node|Rest], Type, Name, DepNode) :-
        find_user_depend(Rest, Type, Name, DepNode).


%%%%%% Add newly-found data dependencies to operation's dependency fields %%%%%

% Add a set of prerequisite constraints to an operation node.
%   arg 1: input:  the node that is to receive the additional constraints
%   arg 2: input:  list of parent nodes that are prerequisites to arg 1
%
add_to_all_prereq(_Node, []) :- !.
add_to_all_prereq(Node, [First | Rest]) :-
        Node == First, !,
        add_to_all_prereq(Node, Rest).  % Node can't be its own prerequisite
add_to_all_prereq(Node, [First | Rest]) :-
        arg(9, First, Children),
        add_to_open_end(Node, Children),
        arg(11, Node, PreParent),
        add_to_open_end(First, PreParent),
        add_to_all_prereq(Node, Rest).

% Add a set of prerequisite constraints to a *list* of operation nodes.
%   arg 1: input:  list of nodes to receive the additional constraints
%   arg 2: input:  list of parent nodes that are prerequisites to arg 1
%
add_list_to_all_prereq([], _Parents) :- !.
add_list_to_all_prereq([Child|Rest], Parents) :-
        add_to_all_prereq(Child, Parents),
        add_list_to_all_prereq(Rest, Parents).

% Add a set of corequisite constraints to an operation node.
%   arg 1: input:  the node that is to receive the additional constraints
%   arg 2: input:  list of parent nodes that are corequisites to arg 1
%
add_to_all_coreq(_Node, []) :- !.
add_to_all_coreq(Node, [First | Rest]) :-
        Node == First, !,
        add_to_all_coreq(Node, Rest).   % Node can't be its own corequisite
add_to_all_coreq(Node, [First | Rest]) :-
        arg(9, First, Children),
        add_to_open_end(Node, Children),
        arg(10, Node, CoParent),
        add_to_open_end(First, CoParent),
        add_to_all_coreq(Node, Rest).
```

```
% Add a set of corequisite constraints to a *list* of operation nodes.
%   arg 1: input:  list of nodes to receive the additional constraints
%   arg 2: input:  list of parent nodes that are corequisites to arg 1
%
add_list_to_all_coreq([], _Parents) :- !.
add_list_to_all_coreq([Child|Rest], Parents) :-
        add_to_all_coreq(Child, Parents),
        add_list_to_all_coreq(Rest, Parents).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% Schedule the operations of a trace into compacted instructions %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Schedule the first operation in an instruction -- also check if this
%  instruction is a "safe" entry point for a program block.
%   arg 1: input:  list of operation nodes ready to be scheduled
%   arg 2: input:  list of operation nodes that will be ready at the next cycle
%   arg 3: input:  list of operation nodes that must end the trace
%   arg 4: input:  pointer to next free instruction in instruction list
%   arg 5: output: pointer to free instruction after end of this trace
%   arg 6: input:  list of branch points yet to be handled
%   arg 7: output: list of branch points to handle after the trace is scheduled
%   arg 8: input:  list of block counts of unscheduled operations along trace
%   arg 9: input:  list of all operations in the current program trace
%   arg 10: input: tree of fix-up operations found to be needed so far
%   arg 11: output: tree of all fix-up operations known after this trace
%
first_schedule_trace(Ready, Later, EndIt, CompLst0, CompLst, BrnLst0, BrnLst,
                [blk(_ABlk,0),blk(BBlk,BCount)|BlkLst0], Trace, FixUp0, FixUp) :- !,
%       ABlk = [ANum|_AType],
%       perm_tree_lookup(ANum, fix(_Safe,FixList,_FixGoto,_UnsafeLab), FixUp0),
%       tie_up_loose_end(FixList),
        BBlk = [BNum|BType],
        check_if_safe(BNum, BType, FixUp0, CompLst0),
        first_schedule_trace(Ready, Later, EndIt, CompLst0, CompLst,
                BrnLst0, BrnLst, [blk(BBlk,BCount)|BlkLst0],
                Trace, FixUp0, FixUp).
first_schedule_trace(Ready, Later, EndIt, CompLst0, CompLst,
                BrnLst0, BrnLst, BlkLst0, Trace, FixUp0, FixUp) :-
        schedule_trace(Ready, Later, EndIt, CompLst0, CompLst,
                BrnLst0, BrnLst, BlkLst0, Trace, FixUp0, FixUp).

% Schedule an operation in the middle of an instruction.
%   [all arguments are the same as for "first_schedule_trace"]
%
schedule_trace([], [], EndIt, [Instr|CompLst0], CompLst, BrnLst, BrnLst,
                BlkL0, _Trace, FixUp, FixUp) :- !,
        Instr = instr(_,_),
        (compatible(EndIt, Instr) ->   % Schedule final operation here
                (schedule_op(EndIt, [Instr|CompLst0], BlkL0, _BlkLst,
                             [], _Rest, [], _Later, FixUp, _Fix1),
                 CompLst0 = CompLst
                ) ;                    % Schedule it during next instruction
                (CompLst0 = [instr(_,_)|_],
                 schedule_trace([], [], EndIt, CompLst0, CompLst,
                        BrnLst, BrnLst, BlkL0, _Trace, FixUp, FixUp)
                )
        ).
schedule_trace([], Later, EndIt, [IDone|CompLst0], CompLst, BrnLst0, BrnLst,
                BlkLst0, Trace, FixUp0, FixUp) :- !,
        IDone = instr(DoneList,_),
        mark_scheduled(DoneList),
        CompLst0 = [instr(NodeList,_)|_],
        schedule_forced_ops(NodeList, CompLst0, BlkLst0, BlkLst1,
                Later, NewReady, [], NewLater, FixUp0, FixUp1),
        first_schedule_trace(NewReady, NewLater, EndIt, CompLst0, CompLst,
                BrnLst0, BrnLst, BlkLst1, Trace, FixUp1, FixUp).
schedule_trace([EndIt|Rest], Later, EndIt, CompLst0, CompLst,
                BrnLst0, BrnLst, BlkLst0, Trace, FixUp0, FixUp) :- !,
                        % OK to throw away "endit" node, we'll get back to it
        schedule_trace(Rest, Later, EndIt, CompLst0, CompLst,
                BrnLst0, BrnLst, BlkLst0, Trace, FixUp0, FixUp).
%
% When a "fixup" operation is encountered, the list of fix-up operations
%   is looked up, the rest of the trace's final dependency status is used
%   to add dependency constraints to the new fix-up operations, and
%   possibly find more operations that are ready to be scheduled; then
%   the scheduling proceeds on the augmented trace.
%
schedule_trace([Now|Rest], Later, _OldEndIt, [Instr|CompLst0], CompLst,
                BrnLst0, BrnLst, BlkLst0, Trace, FixUp0, FixUp) :-
        arg(1, Now, fixup(Num)), !,
%print('fixup '), print(Num), nl,
        perm_tree_store(Num, fix(_Safe,FixList,FixGoto,UnsafeLab), FixUp0),
        tie_up_loose_end(FixList),
        ((FixList == [], arg(1, FixGoto, goto([Dest]))) ->
                ((nonvar(Dest), Dest = [_Indirect]) ->
                        (name(Num, BlkName),
                         append("unsafe", BlkName, UnsafeName),
                         name(UnsafeLab, UnsafeName)
%, print('indirect '), print(Dest), print(' -> '), print(UnsafeLab), nl
                        ) ;
                        UnsafeLab = Dest  % No unsafe ops; bypass extra goto
%, print('direct '), print(Dest), nl )
                ) ;
                (name(Num, BlkName),
                 append("unsafe", BlkName, UnsafeName),
                 name(UnsafeLab, UnsafeName)
%, print('not candidate for direct: '), print(UnsafeLab), nl
                )
        ),
        end_duplicate_op_list(FixList, CopyFix),
        start_block_list(NewBlkLst),
        restart_block_list(BlkLst0, NewBlkLst, BlkLst1),
        list_block_list(BlkLst1, CopyFix, BlkLst2),
        end_duplicate_op_list([FixGoto], [DupGoto]),
        list_block_list(BlkLst2, [DupGoto], BlkLst3),
        end_block_list(BlkLst3, BlkLst4),
        append(Trace, CopyFix, MostTrace),
        append(MostTrace, [DupGoto], ForwTrace),
        add_dependencies(ForwTrace, _, _, [], []),
        reverse(CopyFix, [], BackFix),
        assign_priorities([DupGoto | BackFix], [], Ready, [], After),
        priority_merge_list(Ready, Rest, Rest1),
        priority_merge_list(After, Later, Later1),
        schedule_trace(Rest1, Later1, DupGoto, [Instr|CompLst0], CompLst,
                BrnLst0, BrnLst, BlkLst4, Trace, FixUp0, FixUp).
                        % Note that the next clause does *not* have a cut;
                        % we prefer to schedule a compatible operation now,
                        % but if doing so causes subsequent problems, we
                        % will backtrack and have to schedule it later.
schedule_trace([Now|Rest], Later, EndIt, [Instr|CompLst0], CompLst,
                BrnLst0, BrnLst, BlkLst0, Trace, FixUp0, FixUp) :-
        compatible(Now, Instr),
        arg(1, Now, Op),
        (Op = goto([GoFirst|GoRest]) ->
                                % Schedule goto operation
         (arg(5, Now, TrNum),
          schedule_op(Now, [Instr|CompLst0], BlkLst0, BlkLst1,
                Rest, Rest1, Later, Later1, FixUp0, FixUp1),
                                % Find remaining unscheduled trace
          leftover_trace(Trace, Now, [], BackLeftTrace),
          reverse(BackLeftTrace, [], LeftTrace),
                                % Add unscheduled branches to branch list
          add_branches(GoRest, GoFirst, LeftTrace, _, BrnLst0, BrnLst1),
                                % Schedule subsequent label
          priority_merge_list(Rest1, Later1, Ready1),
          extract_label(GoFirst, FirstDest),
          LabelNode = node(label(FirstDest),0,0,0,TrNum,_,_,_,_,_,_,_,_),
          Ready2 = [LabelNode | Ready1],
          schedule_trace([], Ready2, EndIt, [Instr|CompLst0], CompLst,
                BrnLst1, BrnLst, BlkLst1, Trace, FixUp1, FixUp)
         ) ;
         (schedule_op(Now, [Instr|CompLst0], BlkLst0, BlkLst1,
                Rest, Rest1, Later, Later1, FixUp0, FixUp1),
          schedule_trace(Rest1, Later1, EndIt, [Instr|CompLst0], CompLst,
                BrnLst0, BrnLst, BlkLst1, Trace, FixUp1, FixUp)
         )
         ).
schedule_trace([Now|Rest], Later0, EndIt, CompLst0, CompLst,
                BrnLst0, BrnLst, BlkLst0, Trace, FixUp0, FixUp) :-
% write('*** WARNING in schedule_trace: backtracking to schedule "'),
% arg(1, Now, Op), print(Op), write('" later'), nl,
        priority_merge(Now, Later0, Later1),
        schedule_trace(Rest, Later1, EndIt, CompLst0, CompLst,
                BrnLst0, BrnLst, BlkLst0, Trace, FixUp0, FixUp).

% Schedule operations previously forced into this instruction slot.
%   arg 1: input:  open end list of operations to be scheduled; may be unbound
%   arg 2: input:  current position in the compacted instruction list
%   arg 3: input:  current list of block counts of unscheduled operations
%   arg 4: output: new list of block counts after scheduling this operation
%   arg 5: input:  current list of operations ready to be scheduled
%   arg 6: output: new list of operations ready to be scheduled
%   arg 7: input:  current list of operations schedulable after current instr.
%   arg 8: output: new list of operations schedulable after current instruction
%   arg 9: input:  tree of fix-up operations found to be needed so far
%   arg 10: output: new tree of fix-up operations after scheduling this node
%
schedule_forced_ops(NodeList, CompLst, BlkLst0, BlkLst, Ready0, Ready,
                Later0, Later, FixUp0, FixUp) :-
        nonvar(NodeList), !,
        schedule_op_list(NodeList, CompLst, BlkLst0, BlkLst,
                Ready0, Ready, Later0, Later, FixUp0, FixUp).
schedule_forced_ops(_NodeList, _CompLst, BlkLst, BlkLst, Ready, Ready,
                Later, Later, FixUp, FixUp).


%%%%%%%%%%%% Schedule lists of operations or single operations %%%%%%%%%%%%%

% Schedule a ready operation and find any children that have become ready.
%   arg 1: input:  new operation node being scheduled
%   arg 2: input:  current position in the scheduled instruction list
%   arg 3: input:  current block/type count list
%   arg 4: output: new block count list after these operations are scheduled
%   arg 5: input:  current list of other operations now ready
%   arg 6: output: new list of other operations ready to schedule
%   arg 7: input:  current list of other operations schedulable later
%   arg 8: output: new list of other operations schedulable later
%   arg 9: input:  current tree of fix-up operations
%   arg 10: output: new tree of fix-up operations
%
schedule_op(Node, CompList, B0, BlockList, R0, Rest, L0, Later, F0, FixUp) :-
        schedule_op_instr(Node, CompList),
        schedule_op_child(Node, CompList, B0, BlockList, R0, Rest, L0, Later, F0, FixUp).

% Schedule a ready operation in the current instruction slot.
%   arg 1: input:  the new operation node to schedule here
%   arg 2: input:  the current position in the compacted instruction list
%
schedule_op_instr(Node, CompList) :-
        CompList = [instr(IOps,_)|_CompRest],
        add_to_open_end(Node, IOps),
        arg(7, Node, yes).              % Indicate scheduled position

% Identify the children of a scheduled operation that are now ready.
%   arg 1: input:  new operation node being scheduled
%   arg 2: input:  current position in the scheduled instruction list
%   arg 3: input:  current block/type count list
%   arg 4: output: new block count list after these operations are scheduled
%   arg 5: input:  current list of other operations now ready
%   arg 6: output: new list of other operations ready to schedule
%   arg 7: input:  current list of other operations schedulable later
%   arg 8: output: new list of other operations schedulable later
%   arg 9: input:  current tree of fix-up operations
%   arg 10: output: new tree of fix-up operations
%
schedule_op_child(Node, CompList, B0, BlockList, R0, Rest, L0, Later, F0, FixUp) :-
        arg(2, Node, Block),
        arg(4, Node, Type),
        fix_special([Block|Type], Node, B0, BlockList, F0, FixUp),
        arg(9, Node, Children),
        find_ready_child(Children, RCoreq, RPrereq),
        priority_merge_list(RCoreq, R0, R1),
        priority_merge_list(RPrereq, L0, L1),
        arg(12, Node, Forced),
        force_schedule(Forced, CompList, R1, Rest, L1, Later).

% Find ready children of operations previously forced to be scheduled now.
%   arg 1: input:  open-end list of operations to check for children
%   [other args are the same as those to "schedule_op_child"]
%
schedule_op_list(Var, _CompList, BlockList, BlockList,
                 Rest, Rest, Later, Later, FixUp, FixUp) :-
        var(Var), !.
schedule_op_list([Node|Nodes], CompList, B0, BlockList,
                 R0, Rest, L0, Later, F0, FixUp) :-
        schedule_op_child(Node, CompList, B0, B1, R0, R1, L0, L1, F0, F1),
        schedule_op_list(Nodes, CompList, B1, BlockList,
                         R1, Rest, L1, Later, F1, FixUp).

% Try to schedule forced dependents of an operation now being scheduled.
%   arg 1: input:  forced list of operation nodes
%   arg 2: input:  current position in the compacted instruction list
%   arg 3: input:  list of operation nodes currently known to be ready
%   arg 4: output: new list of operation nodes that are ready to schedule
%   arg 5: input:  list of operation nodes currently schedulable later
%   arg 6: output: new list of operation nodes that are schedulable later
%
force_schedule([], _CompList, Rest, Rest, Later, Later) :- !.
force_schedule([Force|Rest], CompList, R0, Rest, L0, Later) :-
        arg(1, Force, Delay),
        arg(2, Force, Node),
        ensure_ready([Node], [Node]),
        peel_off(Delay, CompList, [DInstr|DRest]),
        compatible(Node, DInstr),
        schedule_op_instr(Node, [DInstr|DRest]),
        force_schedule(Rest, CompList, R0, Rest, L0, Later).


%%%%%% Routines to handle conditional/unconditional goto list elements. %%%%%%

% Extract the label part of a destination.
%   arg 1: input:  one (possibly conditional) goto destination
%   arg 2: output: the label part of that destination
%
extract_label((_Cond,Label), Label) :- !.
extract_label(Label, Label).

% Replace the label part of a destination.
%   arg 1: input:  one (possibly conditional) goto destination
%   arg 2: input:  the new label to be inserted in that destination
%   arg 3: output: updated (possibly conditional) goto destination
%
replace_label((Cond,_Label), NewLabel, (Cond,NewLabel)) :- !.
replace_label(_Label, NewLabel, NewLabel).
```

```
% Extract the conditional part of a destination, or return [] if none.
%   arg 1: input:  one (possibly conditional) goto destination
%   arg 2: output: the conditional part of that destination, or []
%
extract_cond(Var, []) :-
        var(Var), !.            % Goto destination not yet determined
extract_cond((Cond,_Label), Cond) :- !.
extract_cond(_Label, []).       % Unconditional or indirect


% Find the part of the current trace that has already been scheduled.
%   arg 1: input:  list of all operation nodes along the current trace
%   arg 2: input:  operation node of the goto currently being scheduled
%   arg 3: input:  []
%                (reverse list of scheduled trace nodes found so far)
%   arg 4: output: reverse list of all trace nodes that have been scheduled
%
leftover_trace(_List, EndNode, Trace, Trace) :-
        arg(4, EndNode, Type),
        Type < 1, !.                    % No trace if artificial
leftover_trace([Node|_Rest], Node, Trace, Trace) :- !.  % Ends at current goto node
leftover_trace([Node|Rest], EndNode, TrIn, TrOut) :-
        arg(7, Node, Sched),
        nonvar(Sched), !,
        leftover_trace(Rest, EndNode, TrIn, TrOut).
leftover_trace([Node|Rest], EndNode, TrIn, TrOut) :-
        leftover_trace(Rest, EndNode, [Node|TrIn], TrOut).

% At a goto node, add branches yet to be followed to the branch list.
%   arg 1: input:  list of all destinations from the goto operation
%   arg 2: input:  first destination in the goto operation
%   arg 3: input:  list of fix-up operations that must start the new branch
%   arg 4: input:  branch list up to this point
%   arg 5: output: complete branch list including the new branches
%
add_branches([], _GoFirst, _Trace, NewBrn, BrnLst0, BrnLst) :- !,
        tie_up_loose_end(NewBrn),
        append(NewBrn, BrnLst0, BrnLst).
add_branches([Go|GoRest], GoFirst, Trace, NewBrn, BrnLst0, BrnLst) :-
        extract_label(Go, GoLabel),
        extract_label(GoFirst, GoFirstLabel),
        (GoLabel = GoFirstLabel ->
                true ;          % Don't duplicate first destination block
                (unreplace_label(GoLabel, GoToLabel),
                 add_to_open_end(br(Trace, GoToLabel, GoLabel), NewBrn)
                )
        ),
        add_branches(GoRest, GoFirst, Trace, NewBrn, BrnLst0, BrnLst).

% Mark a list of nodes that have been previously scheduled.
%   arg 1: input:  list of nodes scheduled in the previous instruction slot
%
mark_scheduled([]) :- !.
mark_scheduled([Node|Rest]) :-
        arg(8, Node, yes),
        mark_scheduled(Rest).
```

```
%%%%%%%%%%% Detect new operations that have become ready to execute %%%%%%%%%%%

% From a list of possible nodes, find those that are now ready.
%   arg 1: input:  list (maybe open-ended) of operation nodes to check
%   arg 2: output: list of operation nodes ready to be scheduled now
%   arg 3: output: list of operation nodes ready to schedule later
%
find_ready_child(Var, [], []) :-
        var(Var), !.
find_ready_child([], [], []) :- !.
find_ready_child([Node | Rest], Now, Later) :-
        arg(10, Node, CoReq),
        arg(11, Node, PreReq),
        (ready_now(CoReq) ->
                (ready_before(PreReq) ->
                        (Now = [Node | NowRest],
                         Later = LaterRest
                        );
                        (ready_now(PreReq) ->
                                (Now = NowRest,
                                 Later = [Node | LaterRest]
                                );
                                (Now = NowRest,
                                 Later = LaterRest
                                )
                        )
                );
                (Now = NowRest,
                 Later = LaterRest
                )
        ),
        find_ready_child(Rest, NowRest, LaterRest).

% Detect if all operation nodes in a list have been scheduled as of now.
%   arg 1: input:  list of operation nodes to check
%
ready_now(Var) :-
        var(Var), !.
ready_now([]) :- !.
ready_now([Node | Rest]) :-
        arg(7, Node, Sched),
        nonvar(Sched),
        ready_now(Rest).

% Detect if all operation nodes in a list have previously been scheduled.
%   arg 1: input:  list of operation nodes to check
%
ready_before(Var) :-
        var(Var), !.
ready_before([]) :- !.
ready_before([Node | Rest]) :-
        arg(8, Node, Sched),
        nonvar(Sched),
        ready_before(Rest).
```

```
% Block list routines identify when operations are scheduled out-of-order.
% This information is used to make sure all the right operations are
%   performed if a block is entered from a different place.
% The ground rules are:
%   * if a block flows into another (i.e., ended w/unconditional goto), then
%     + it has a "safe" entry point if there is an instruction before which
%       *all* previous blocks' operations have been scheduled, but *no*
%       later blocks' operations have been scheduled --- in this case, the
%       fixup list contains all operations from the block that were
%       scheduled before the safe entry point, and "goto safe<thisblock>"
%     + otherwise (no safe entry point), the fixup list contains *all*
%       operations from this block, and "goto unsafe<nextblock>".
%   * otherwise (the block ends with a conditional goto), then
%     + the same rules as above apply for finding a "safe" entry point
%       (except that the conditional goto also must not have been scheduled)
%     + otherwise (no safe entry point), the fixup list contains *all*
%       operations from this block, and the final "goto" instruction
%       with *all* destinations made unsafe.
% The basic algorithm is  (using fix(Safe,FixList,FixGoto,Lab) structures):
%   * Bind "Safe" to "safe" when (and if) a safe entry point is found --
%     all previous blocks must be scheduled and "FixGoto" still unbound;
%     "FixGoto" is also bound to go to the safe entry point for this block.
%   * Always decrement appropriate count in the block list.
%   * If current op matches head of block list and this block is "safe",
%     do nothing else; otherwise...
%   * If current op does not match head of block list, scan down block
%     list until current block is found; for each block along the way, if
%     it is not already "safe" (this really only applies to the head of
%     the block list) and does not already have a "FixGoto", bind
%     its "FixGoto" to go to the (unsafe) next block in the block list.
%   * If current op is a "goto", bind its block's "FixGoto" to the
%     unsafe version of the goto list.
%   * If current op is not a "goto", add it to its block's "FixList".
%   * However, do not lookup or bind fix() entries for copied blocks
%     (type -1 or -2), and do *nothing* for artificial (type 0) blocks.

% Update block list for newly-scheduled operation; also, if necessary,
%  add the operation to the appropriate fix-up lists.
%   arg 1: input:  block number/type of the current operation
%   arg 2: input:  the operation node being scheduled
%   arg 3: input:  the current block list
%   arg 4: output: the new block list after this operation is scheduled
%   arg 5: input:  the current tree of fix-up operations
%   arg 6: output: the new tree of needed fix-up operations
%
                        % Artificial nodes don't affect block lists
fix_special([0|0], _Node, BlkLst, BlkLst, FixUp, FixUp) :- !.
fix_special(BlockType, Node, BlkLst0, BlkLst, FixUp, FixUp) :-
        update_blocks(BlockType, Node, BlkLst0, BlkLst),
        BlkLst = [blk(LBlockType,_LCount)|_],
        BlockType = [Block|Type],
        perm_tree_store(Block, fix(Safe,FixList,FixGoto,_UnsafeLab), FixUp),
        ((BlockType == LBlockType, nonvar(Safe)) ->
            true ;
            (BlockType == LBlockType ->
                true ;
                (scan_for_unsafe(BlkLst, BlockType, FixUp)
                )
            ),
            (Type =:= 1 ->
                (arg(1, Node, goto(List)) ->
                    (List == [[_Indirect]] ->
                        add_to_open_end(Node, FixList) ;
                        (Node = node(_,BLK,IDX,TYP,TR,_,_,_,_,_,_,_,USR,ULAB),
                         unsafe_replace_labels(List, FixUp, NewList),
                         FixGoto = node(goto(NewList),BLK,IDX,TYP,TR,
                                        _,_,_,_,_,_,_,USR,ULAB)
                        )
                    ) ;
                    add_to_open_end(Node, FixList)
                ) ;
                true
            )
        ).

% Remove current instruction from the (unscheduled) block list structure.
%   arg 1: input:  block number and type of the current instruction
%   arg 2: input:  pointer to the current instruction node
%   arg 3: input:  previous block list structure
%   arg 4: output: new block list structure, less the current node
%
update_blocks(BlockType, _Node, [], []) :- !,
        write('*** WARNING: block/type '), write(BlockType),
        write(' is not included in block list'), nl.
update_blocks(BlockType, _Node, [blk(BlockType,Count)|BL],
                        [blk(BlockType,NewCount)|BL]) :- !,
        NewCount is Count - 1.
update_blocks(BlockType, Node, [blk(OBlockType,Count)|Rest],
                        [blk(OBlockType,Count)|BL]) :-
        update_blocks(BlockType, Node, Rest, BL).

% Check if the scheduling of a new operation makes any other blocks "unsafe";
%  if so, mark the fixup lists of those blocks.
%   arg 1: input:  current block list structure
%   arg 2: input:  block number/type of the new operation
%   arg 3: input:  current fixup list structure
%
scan_for_unsafe([blk(_,_)], BlockType, _FixUp) :- !,
        write('*** WARNING: block/type '), write(BlockType),
        write(' was not included in block list'), nl.
scan_for_unsafe([blk(BlockTypeA,_CountA),blk(BlockTypeB,CountB)|BL], BlockType, FixUp) :-
        BlockTypeA = [BlockA|TypeA],
        (TypeA =:= 1 ->
                (perm_tree_store(BlockA, fix(Safe,_FixList,FixGoto,_Lab), FixUp),
                 ((var(Safe), var(FixGoto)) ->
                        (BlockTypeB = [BlockB|_TypeB],
                         unsafe_add_label(BlockB, UnsafeB),
                         FixGoto = node(goto([UnsafeB]),BlockA,0,0,_,_,_,_,_,_,_,_,_,_)
                        ) ;
                        true
```

```
                  )
                ) ;
                true
        ),
        (BlockTypeB == BlockType ->
                true ;
                scan_for_unsafe([blk(BlockTypeB,CountB)|BL], BlockType, FixUp)
        ).

% Schedule "safe" label here if this is a safe entry point.
%   arg 1: input:  current instruction's block number
%   arg 2: input:  current instruction's type
%   arg 3: input:  current fixup list structure
%   arg 4: input:  current position in compacted instruction list
%
check_if_safe(Block, 1, FixUp, CompLst) :-        % Only "real" nodes can be safe
        perm_tree_store(Block, fix(Safe,_FixList,FixGoto,_Lab), FixUp),
        var(FixGoto), !,                   % Safe only if no goto is already here
        Safe = safe,
        name(Block, BName),
        append("safe", BName, SafeName),
        name(SafeNum, SafeName),
        FixGoto = node(goto([SafeNum]),0,0,0,_,_,_,_,_,_,_,_,_),
        LabelNode = node(label(SafeNum),0,0,0,_,_,_,_,_,_,_,_,_),
        schedule_op_instr(LabelNode, CompLst).
check_if_safe(_Block, _Type, _FixUp, _CompLst). % Nope, not safe


% Duplicate an operation list, changing the nodes' types to indicate that
%  the nodes are "leftover" nodes, not needing to be fixed up.
%   arg 1: input:  list of operation nodes to be duplicated
%   arg 2: output: list of duplicated operation nodes
%
duplicate_op_list([], []) :- !.
duplicate_op_list([First | Rest], [NewFirst | NewRest]) :-
        First = node(OP,BLK,IDX,_,_,_,_,_,_,_,_,USR,ULAB),
        NewFirst = node(OP,BLK,IDX,-1,_,_,_,_,_,_,_,USR,ULAB),
        duplicate_op_list(Rest, NewRest).

% Duplicate an operation list, changing the nodes' types to indicate that
%  the nodes are "fixup" nodes, not subject to further "fixing up".
%   arg 1: input:  list of operation nodes to be duplicated
%   arg 2: output: list of duplicated operation nodes
%
end_duplicate_op_list([], []) :- !.
end_duplicate_op_list([First | Rest], [NewFirst | NewRest]) :-
        First = node(OP,BLK,IDX,_,_,_,_,_,_,_,_,USR,ULAB),
        NewFirst = node(OP,BLK,IDX,-2,_,_,_,_,_,_,_,USR,ULAB),
        end_duplicate_op_list(Rest, NewRest).


% Extract the completed microinstructions from list of compacted instructions.
%   arg 1: input:  compacted instruction list from trace scheduler
%   arg 2: output: pure instruction list, in form readable by simulator
%
extract_instrs([], []) :- !.
extract_instrs([instr(NList,ResUse)|Rest], [IList|IRest]) :-
        extract_instr(NList, ResUse, IList),
        extract_instrs(Rest, IRest).

% Add resource usage info to one instruction's list of operation nodes.
%   arg 1: input:  instruction's list of operations
%   arg 2: input:  instruction's resource usage structure
%   arg 3: output: combined operation list and resource usage info
%
extract_instr([], ResUse, [{ResCount}]) :- !,
        extract_counts(ResUse, 1, ResCount).
extract_instr([Node|Rest], ResUse, [Op|ORest]) :-
        arg(1, Node, Op),
        extract_instr(Rest, ResUse, ORest).

% Count the number of each type of resource that was used.
%   arg 1: input:  resource usage structure to be counted (can have any arity)
%   arg 2: input:  1
%              (tracks the next member of resource structure to examine)
%   arg 3: output: structure giving usage count for every tracked resource
%
extract_counts(ResUse, Arg, Counts) :-
        arg(Arg, ResUse, List),
        bound_count(List, 0, Count),
        NextArg is Arg + 1,
        (arg(NextArg, ResUse, _Something) ->
                (Counts = (Count,Rest),
                 extract_counts(ResUse, NextArg, Rest)
                ) ;
                Counts = Count
        ).


% Count the total number of elements in a list.
%   arg 1: input:  the list to be counted
%   arg 2: input:  0
%              (running sum of elements counted so far)
%   arg 3: output: the total number of elements
%
count([], Count, Count) :- !.
count([_|Rest], C0, Count) :-
        C1 is C0 + 1,
        count(Rest, C1, Count).

% Count the number of bound-to-non-nil elements in a list.
%   arg 1: input:  the list to be counted
%   arg 2: input:  0
%              (running sum of elements counted so far)
%   arg 3: output: the total number of bound elements
%
bound_count([], Count, Count) :- !.
bound_count([[]|Rest], C0, Count) :- !,         % Nil or unbound
        bound_count(Rest, C0, Count).
bound_count([_|Rest], C0, Count) :-
        C1 is C0 + 1,
        bound_count(Rest, C1, Count).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%% Printing subroutines %%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- dynamic nodecount/1.
nodecount(0).

% Print an operation node without causing infinite recursion!
%   arg 1: input:  the operation node to print
%
portray(node(OP,BLK,IDX,_,_,_,_,_,_,_,_,_,_,_)) :- !,
        write('node('), writeq(BLK), write('-'), writeq(IDX),
        write(':'), writeq(OP), write(')').
%portray(node(OP,BLK,IDX,_TR,_TYP,_PRI,_SCH,_PSCH,_CHLD,_COR,_PRE,_FOR,_USR,_ULB)) :-
%       nodecount(1), !,
%       write('{'), write(BLK), write('-'), write(IDX),
%       write(':'), write(OP), write('}').
%portray(node(OP,BLK,IDX,TR,TYP,PRI,SCH,PSCH,CHLD,COR,PRE,FOR,USR,ULB)) :- !,
%       assert(nodecount(1)),
%       write('node('), print(OP), write(','),
%       print(BLK), write(','), print(IDX), write(','),
%       print(TR), write(','), print(TYP), write(','),
%       print(PRI), write(','), print(SCH), write(','),
%       print(PSCH), write(','), print(CHLD), write(','),
%       print(COR), write(','), print(PRE), write(','),
%       print(FOR), write(','), print(USR), write(','),
%       print(ULB), write(')'),
%       retract(nodecount(1)).
%portray(SomethingElse) :-
%       atomic(SomethingElse),
%       writeq(SomethingElse).


% Pretty-print a (possibly nested) list, one (indented) element per line.
%   arg 1: input:  the list to be pretty-printed
%
writelst(List) :- writelst(List, 0).

writelst([First | Rest], Num) :-
    !,
    tab(Num), write('['), nl,
    NewNum is Num+1,
    writesub([First | Rest], NewNum).
writelst(NonList, Num) :-
    tab(Num), writeq(NonList).

writesub([], Num) :- !,
    NewNum is Num-1,
    tab(NewNum), write(']').
writesub([Only], Num) :- !,
    writelst(Only, Num), nl,
    writesub([], Num).
writesub([First | Rest], Num) :-
    writelst(First, Num), write(','), nl,
    writesub(Rest, Num).
```

```
%%%%%%%%%%%%%%%%%%% Routines to deal with open-ended lists %%%%%%%%%%%%%%%%%%

% Add an element to an open end list.
%    arg 1: input:  the element to be added
%    arg 2: input:  the open end list to receive the new element
%
add_to_open_end(Element, [Element|_]) :- !.
add_to_open_end(Element, [_|Rest]) :-
    add_to_open_end(Element, Rest).

% Add a *list* of elements to an open end list.
%    arg 1: input:  list of elements to be added
%    arg 2: input:  the open end list to receive the new elements
%
add_list_to_open_end([], _) :- !.
add_list_to_open_end([First|Rest], List) :-
    add_to_open_end(First, List),
    add_list_to_open_end(Rest, List).

% Peel off a given number of elements from head of an open-ended list.
%    arg 1: input:  number of elements to peel off
%    arg 2: input:  open ended list to be peeled
%    arg 3: output: pointer into open ended list after doing the peeling
%
peel_off(0, List, List) :- !.
peel_off(Num, [_|L0], List) :-
        NewNum is Num-1,
        peel_off(NewNum, L0, List).

% Tie up the end of an open-end list, so that it cannot get any more elements.
%    arg 1: input:  the open end list to be finished off
%
tie_up_loose_end([]) :- !.
tie_up_loose_end([_|Rest]) :-
    tie_up_loose_end(Rest).


%%%%%%%%%%%%%%%%%% Routines to handle binary search trees %%%%%%%%%%%%%%%%%%
% each node of the tree has the form  node(Name, Value, LeftChild, RightChild)

%%%% The first set of routines deals with "permanent" trees, in which each
%%%%  name is associated with only one value for the duration of the program

% Lookup a value in a tree; print error if given name is not already in tree.
%    arg 1: input:  the name to be looked up
%    arg 2: output: the value assigned to that name
%    arg 3: input:  the permanent tree structure
%
perm_tree_lookup(Name, 0, Var) :-
    var(Var), !,
    write('*** Error: can''t look up value of '), print(Name),
    write(' (using the value 0 instead) ***'), nl.
perm_tree_lookup(Name, Value, node(Name, Value, _Left, _Right)) :- !.
perm_tree_lookup(Name, Value, node(CheckName, _CheckValue, Left, _Right)) :-
    Name @< CheckName, !,
    perm_tree_lookup(Name, Value, Left).
perm_tree_lookup(Name, Value, node(CheckName, _CheckValue, _Left, Right)) :-
    Name @> CheckName, !,
    perm_tree_lookup(Name, Value, Right).

% Lookup or store a value in a tree; create a new node if the given name was
%  not already in the tree.
%  [arguments are the same as for "perm_tree_lookup"]
%
perm_tree_store(Name, Value, node(Name, Value, _Left, _Right)) :- !.
perm_tree_store(Name, Value, node(CheckName, _CheckValue, Left, _Right)) :-
    Name @< CheckName, !,
    perm_tree_store(Name, Value, Left).
perm_tree_store(Name, Value, node(CheckName, _CheckValue, _Left, Right)) :-
    Name @> CheckName, !,
    perm_tree_store(Name, Value, Right).

% Lookup or store a *list* of values in a tree.
%   arg 1: input:  list of names to be looked up
%   arg 2: output: list of values assigned to the names
%   arg 3: input:  the permanent tree structure
%
perm_tree_store_list([], [], _Tree) :- !.
perm_tree_store_list([First|Rest], [LFirst|LRest], Tree) :-
        perm_tree_store(First, LFirst, Tree),
        LFirst = node(_,_,_,_,_,_,_,_,_,_,_,_,_,_),
        perm_tree_store_list(Rest, LRest, Tree).

% Lookup a list of 'forced' nodes in a tree; same as 'perm_tree_store_list'
%       except that the input list contains *pairs* of entries, one of which
%       is a node name to be translated, and the other to remain unchanged.
%   arg 1: input:  list of node name/delay value pairs to be looked up
%   arg 2: output: result list of nodes/delay value pairs
%   arg 3: input:  the permanent tree structure
%
perm_tree_store_forced([], [], _Tree) :- !.
perm_tree_store_forced([(First,Delay)|Rest], [(LFirst,Delay)|LRest], Tree) :-
        perm_tree_store(First, LFirst, Tree),
        LFirst = node(_,_,_,_,_,_,_,_,_,_,_,_,_,_),
        perm_tree_store_forced(Rest, LRest, Tree).


%%%% The second set of routines deals with "temporary" trees, in which each
%%%%  name may potentially take on many different values

% Lookup a value in a tree; return [] if given name is not already in tree.
%   arg 1: input:  the name to be looked up
%   arg 2: output: the value assigned to that name
%   arg 3: input:  the temporary tree structure
%
temp_tree_lookup(_Name, [], []) :- !.   % Return '[]' if unknown
temp_tree_lookup(Name, Value, node(Name, Value, _Left, _Right)) :- !.
temp_tree_lookup(Name, Value, node(CheckName, _CheckValue, Left, _Right)) :-
    Name @< CheckName, !,
    temp_tree_lookup(Name, Value, Left).
temp_tree_lookup(Name, Value, node(CheckName, _CheckValue, _Left, Right)) :-
```

```
    Name @> CheckName,
    temp_tree_lookup(Name, Value, Right).

% Store a value in a tree; replace any previously stored value.
%   arg 1: input:  the name to be assigned a value
%   arg 2: output: the value to assign to that name
%   arg 3: input:  previous temporary tree structure
%   arg 4: output: new temporary tree structure, with modified value
%
temp_tree_store(Name, Value, [], node(Name, Value, [], [])) :- !.
temp_tree_store(Name, Value, node(Name, _OldValue, Left, Right),
                        node(Name, Value, Left, Right)) :- !.
temp_tree_store(Name, Value, node(CheckName, CheckValue, Left, Right), NewNode) :-
    Name @< CheckName, !,
    NewNode = node(CheckName, CheckValue, NewLeft, Right),
    temp_tree_store(Name, Value, Left, NewLeft).
temp_tree_store(Name, Value, node(CheckName, CheckValue, Left, Right), NewNode) :-
    Name @> CheckName, !,
    NewNode = node(CheckName, CheckValue, Left, NewRight),
    temp_tree_store(Name, Value, Right, NewRight).


% Variant of temp_tree_lookup, that handles "two-level" memory reference trees.
%   arg 1: input:  the register or memory location to be looked up
%   arg 2: output: the value assigned to that name
%   arg 3: input:  the temporary tree structure
%
temp_tree_var_lookup(mem(BaseReg,Offset), Value, Tree) :- !,
        temp_tree_lookup(mem(BaseReg), SubTree, Tree),
        temp_tree_lookup(Offset, Value, SubTree).
temp_tree_var_lookup(Reg, Value, Tree) :-
        temp_tree_lookup(Reg, Value, Tree).

% Variant of temp_tree_lookup, that handles "two-level" memory reference trees.
%   arg 1: input:  the register or memory location to be modified
%   arg 2: input:  the value to assign to that name
%   arg 3: input:  original temporary tree structure
%   arg 4: output: modified temporary tree structure, with new assignment
%
temp_tree_var_store(mem(BaseReg,Offset), Value, TreeIn, TreeOut) :- !,
        temp_tree_lookup(mem(BaseReg), SubTreeIn, TreeIn),
        temp_tree_store(Offset, Value, SubTreeIn, SubTreeOut),
        temp_tree_store(mem(BaseReg), SubTreeOut, TreeIn, TreeOut).
temp_tree_var_store(Reg, Value, TreeIn, TreeOut) :-
        temp_tree_store(Reg, Value, TreeIn, TreeOut).


%%%%%%%%%%%%%%%%%%%%%%%%%% Miscellaneous subroutines %%%%%%%%%%%%%%%%%%%%%%%%

% The ubiquitous list append/concat routine.
%   arg 3 is the concatenation of arg 1 and arg 2
%
append([], List, List) :- !.
append([First | Rest], List, [First | New]) :-
        append(Rest, List, New).
```

```
% The almost-as-ubiquitous list reversing routine.
%   arg 1: input:  the list to be reversed
%   arg 2: input:  []
%                (holds intermediate reversed lists)
%   arg 3: output: the reversed list
%
reverse([], Out, Out) :- !.
reverse([First | Rest], In, Out) :-
        reverse(Rest, [First | In], Out).

% Find the larger value of two numbers.
%   arg 1 and arg 2: inputs:  the numbers to be compared
%   arg 3: output:  the larger of arg 1 and arg 2
%
max(A, B, A) :- A > B, !.
max(A, B, B) :- B >= A.
```

A.2.2 Auxiliary Resource-Limitation File

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% Resource usage monitoring, limiting, and displaying %%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Print a header message telling about tracked resources and limits.
%
write_usage_info :-
        write('{alu=one+disp, bus=two, memread=double, memwrite=double}').

% Determine the limits on resource usage during any single instruction.
%
                                % Limit of 1 alu and 2 buses
usage_avail(avail([_], [_,_], _, _)).

maxmemref(2).                   % Limit of double memory references


% Prevent any further usage of one type of resource.
%   arg 1: input:  one element of a resource usage structure to be locked out
%
prevent_usage([]) :- !.         % (may tie up an open-ended list)
prevent_usage([[]|Rest]) :- !,
        prevent_usage(Rest).
prevent_usage([_|Rest]) :-
        prevent_usage(Rest).


% Test if an operation node is compatible with other operations.
%   arg 1: input:  new node we're trying to schedule now
%   arg 2: input:  instruction structure of other operations in this slot
compatible(Node, Instr) :-
        arg(1, Node, Op),
        arg(2, Instr, ResUse),
        usage_avail(ResUse),    % make sure resource usage structure is set
        find_op_usage(Op, ResUse), !.
compatible(Node, Instr) :-
        arg(1, Instr, IList),
        var(IList), !,          % no previous ops in this instruction
        arg(1, Node, Op),
        write('*** ERROR in compatible: '),
        write(Op), write(' cannot be scheduled'), nl, abort.

% Find the resource usage of one microoperation; if the resource usage has
%  been limited, and this operation goes over a limit, this routine will fail.
%   arg 1: input:  operation to be checked
%   arg 2: input:  usage structure to (possibly) be augmented
%
find_op_usage((Dest <-- Src), Usage) :- !,
        arg(2, Usage, Bus),
        add_to_open_end((Dest <-- Src), Bus),
        find_src(Src, Usage),
        find_dest(Dest, Usage).
find_op_usage(goto(List), Usage) :- !,
        find_goto(List, Usage).
```

```
find_op_usage(label(_), _) :- !.
find_op_usage(set(_), _) :- !.
find_op_usage(show(_), _) :- !.
find_op_usage(fixup(_), _) :- !.
find_op_usage(end, _) :- !.
find_op_usage(loadlabel(Dest,_), Usage) :- !,
        find_dest(Dest, Usage).
find_op_usage(Unknown, _) :-
        write('*** WARNING: unknown op '), write(Unknown), nl.

% Find the resources used by the source of a data transfer operation.
%   arg 1: input:  source to be checked
%   arg 2: input:  usage structure to (possibly) be augmented
%
find_src((_Tag@Val), Usage) :- !,
        find_src(Val, Usage).
find_src(type(Val), Usage) :- !,
        find_src(Val, Usage).
find_src(value(Val), Usage) :- !,
        find_src(Val, Usage).
find_src(mem(Addr), Usage) :- !,
        arg(3, Usage, Read),
        arg(4, Usage, Write),
        find_memory(Addr, Read, Write).
%       find_src(Addr, Usage).          %% No ALU used for displacement calcs.
find_src(Op, Usage) :-
        functor(Op, _, 1), !,
        arg(1, Usage, Alu),
        add_to_open_end(Op, Alu),
        arg(1, Op, Arg1),
        find_src(Arg1, Usage).
                %% Lump all tag comparisons into the same 'ALU operation'
find_src((Val ? _Tag), Usage) :- !,
        arg(1, Usage, Alu),
        add_to_open_end((Val ? tag), Alu),
        find_src(Val, Usage).
find_src((Val \? _Tag), Usage) :- !,
        arg(1, Usage, Alu),
        add_to_open_end((Val ? tag), Alu),
        find_src(Val, Usage).
find_src((Funct/Arg), _) :-     % Structure name is a constant
        atom(Funct),
        number(Arg), !.
find_src(Op, Usage) :-
        functor(Op, _, 2), !,
        arg(1, Usage, Alu),
        add_to_open_end(Op, Alu),
        arg(1, Op, Arg1),
        arg(2, Op, Arg2),
        find_src(Arg1, Usage),
        find_src(Arg2, Usage).
find_src(_, _).                 % Presumably, something atomic

% Find the resources used by the destination of a data transfer operation.
%   arg 1: input:  destination to be checked
%   arg 2: input:  usage structure to (possibly) be augmented
```

```
%
find_dest(mem(Addr), Usage) :- !,
        arg(4, Usage, Write),
        arg(3, Usage, Read),
        find_memory(Addr, Write, Read).
%       find_src(Addr, Usage).          %% No ALU used for displacement calcs.
find_dest(_, _).                % Presumably, something atomic

% Find the resources used by a goto operation's destination list.
%   arg 1: input:  destination list for the goto operation
%   arg 2: input:  usage structure to (possibly) be augmented
%
find_goto([], _) :- !.
find_goto([First|Rest], Usage) :-
        extract_cond(First, Cond),
        find_src(Cond, Usage),
        find_goto(Rest, Usage).

% Find an operation's memory resource usage.
%   arg 1: input:  the memory address currently being accessed (read/written)
%   arg 2: input:  memory read or write usage structure (matches current
%                  memory operation: read or write)
%   arg 3: input:  memory write or read usage structure -- always the opposite
%                  of arg 2
find_memory(Addr, ThisUse, OppUse) :-
        addr_use(Addr, [], BaseReg, +, 0, Offset),
        add_to_open_end(mem(BaseReg,Offset), ThisUse),
        prevent_usage(OppUse),
        memory_compatible(ThisUse).

% Check if all concurrent memory accesses are compatible with each other.
%   arg 1: input:  the memory access resource usage structure to check
%
memory_compatible(Var) :-
        var(Var), !.
memory_compatible([]) :- !.
memory_compatible([mem(BaseReg,Offset)|Rest]) :-
        memory_compatible(Rest, BaseReg, Offset).

memory_compatible(Var, _, _) :-
        var(Var), !.
memory_compatible([], _, _) :- !.
memory_compatible([mem(BaseReg2,Offset2)|Rest], BaseReg, Offset) :-
        BaseReg2 == BaseReg,
        maxmemref(MaxMemRef),
        MemMask is MaxMemRef - 1,
        CheckOff is Offset \/ MemMask,
        CheckOff2 is Offset2 \/ MemMask,
        CheckOff == CheckOff2,
        memory_compatible(Rest, BaseReg, Offset).
```

A.3 Instruction Pre-Simulator

```
/***************************************************************************
 *  Pre-simulator to add resource usage info to hand-microcoded programs  *
 *                       written by Richard Carlson                       *
 *                          version 2.1    5-18-89                        *
 ***************************************************************************/

/*     Copyright 1988,89  The Regents of the University of California      */
/*                           All Rights Reserved                           */


:- op( 950, xfx, <--). % destructive assignment
:- op( 700, xfx, \=).  % test for value inequality
:- op( 700, xfx, ?).   % test for type equality
:- op( 700, xfx, \?).  % test for type inequality
:- op( 690, fy, ^).    % "pointer to"
:- op( 400, fy, ~).    % bitwise complement operator
:- op( 300, fx, ^^).   % "unbound variable"
:- op( 250, xfx, @).   % concatenation of type and value fields



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%% Overall top-level pre-simulation predicates %%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Top-level call to pre-simulate a program.
%   arg 1: input:  filename of program to pre-simulate (without .hand extension)
%
presim(InName) :-
        make_in_name(InName, NewInName),
        see(NewInName),
        read(Prog),
        seen,
        add_resource_usage(Prog, NewProg),
        name(InName, InList),
        convert_name(InList, OutList),
        name(OutName, OutList),
        tell(OutName),
        write_usage_info,
        write('.'), nl,
        writelst(NewProg),
        write('.'), nl,
        told.

% Convert a base filename into the filename of the input microcode.
%   arg 1: input:  the base filename (with no extension)
%   arg 2: output: the filename with ".hand" extension
%
make_in_name(InName, NewInName) :-
        name(InName, InList),
        name('.hand', InSuffix),
        append(InList, InSuffix, NewInList),
        name(NewInName, NewInList).
```

```
% Convert an input filename into an output filename.
%   arg 1: input:  the input filename (with or without ".hand" extension)
%   arg 2: output: the output filename (with ".compact" extension)
%
convert_name(InSuffix, OutSuffix) :-
        name('.hand', InSuffix), !,
        name('.compact', OutSuffix).
convert_name([], OutSuffix) :- !,
        name('.compact', OutSuffix).
convert_name([Char | InRest], [Char | OutRest]) :-
        convert_name(InRest, OutRest).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% Resource usage monitoring, limiting, and displaying %%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Print a header message telling about tracked resources and limits.
%
write_usage_info :-
        write('{alu=nolim, bus=nolim, memread=double, memwrite=double}').

% Determine the limits on resource usage during any single instruction.
%
                                % Limits only on memory references
usage_avail(avail(_, _, [_,_], [_,_])).

% Prevent any further usage of one type of resource.
%   arg 1: input:  one element of a resource usage structure to be locked out
%
prevent_usage([]) :- !.         % (may tie up an open-ended list)
prevent_usage([[]|Rest]) :- !,
        prevent_usage(Rest).
prevent_usage([_|Rest]) :-
        prevent_usage(Rest).


% Add resource usage information to each instruction.
%
add_resource_usage([], []) :- !.
add_resource_usage([[FirstList]|Rest], [NewFirst|NewRest]) :- !,
        track_ops([FirstList], ResUse),
        extract_counts(ResUse, 1, Counts),
        append([FirstList], [{Counts}], NewFirst),
        add_resource_usage(Rest, NewRest).
add_resource_usage([Single|Rest], NewInstr) :-
        add_resource_usage([[Single]|Rest], NewInstr).

% Track the resource usage for all operations in an instruction slot
%   arg 1: input:  list of new operations for this instruction slot
%   arg 2: input:  resource usage structure of this instruction slot
track_ops([], _) :- !.
track_ops([Op|Rest], ResUse) :-
        usage_avail(ResUse),    % make sure resource usage structure is set
        find_op_usage(Op, ResUse), !,
        track_ops(Rest, ResUse).
track_ops(BadList, _ResUse) :-
        write('*** ERROR: operation list '), write(BadList),
        write(' cannot be scheduled'), nl.

% Find the resource usage of one microoperation; if the resource usage has
%  been limited, and this operation goes over a limit, this routine will fail.
%   arg 1: input:  operation to be checked
%   arg 2: input:  usage structure to (possibly) be augmented
%
find_op_usage((Dest <-- Src), Usage) :- !,
        arg(2, Usage, Bus),
        add_to_open_end((Dest <-- Src), Bus),
        find_src(Src, Usage),
        find_dest(Dest, Usage).
find_op_usage(goto(List), Usage) :- !,
        find_goto(List, Usage).
find_op_usage(_, _).

% Find the resources used by the source of a data transfer operation.
%   arg 1: input:  source to be checked
%   arg 2: input:  usage structure to (possibly) be augmented
%
find_src((_Tag@Val), Usage) :- !,
        find_src(Val, Usage).
find_src(mem(Addr), Usage) :- !,
        arg(3, Usage, Read),
        arg(4, Usage, Write),
        find_memory(Addr, Read, Write),
        find_src(Addr, Usage).
find_src(Op, Usage) :-
        functor(Op, _, 1), !,
        arg(1, Usage, Alu),
        add_to_open_end(Op, Alu),
        arg(1, Op, Arg1),
        find_src(Arg1, Usage).
find_src(Op, Usage) :-
        functor(Op, _, 2), !,
        arg(1, Usage, Alu),
        add_to_open_end(Op, Alu),
        arg(1, Op, Arg1),
        arg(2, Op, Arg2),
        find_src(Arg1, Usage),
        find_src(Arg2, Usage).
find_src(_, _).                 % Presumably, something atomic

% Find the resources used by the destination of a data transfer operation.
%   arg 1: input:  destination to be checked
%   arg 2: input:  usage structure to (possibly) be augmented
%
find_dest(mem(Addr), Usage) :- !,
        arg(4, Usage, Write),
        arg(3, Usage, Read),
        find_memory(Addr, Write, Read),
        find_src(Addr, Usage).
find_dest(_, _).                % Presumably, something atomic
```

```
% Find the resources used by a goto operation's destination list.
%   arg 1: input:  destination list for the goto operation
%   arg 2: input:  usage structure to (possibly) be augmented
%
find_goto([], _) :- !.
find_goto([First|Rest], Usage) :-
        extract_cond(First, Cond),
        find_src(Cond, Usage),
        find_goto(Rest, Usage).

% Extract the conditional part of a destination, or return [] if none.
%   arg 1: input:  one (possibly conditional) goto destination
%   arg 2: output: the conditional part of that destination, or []
%
extract_cond((Cond,_Label), Cond) :- !.
extract_cond(_Label, []).

% Find an operation's memory resource usage.
%   arg 1: input:  the memory address currently being accessed (read/written)
%   arg 2: input:  memory read or write usage structure (matches current
%                  memory operation: read or write)
%   arg 3: input:  memory write or read usage structure -- always the opposite
%                  of arg 2
find_memory(Addr, ThisUse, OppUse) :-
        addr_use(Addr, [], BaseReg, +, 0, Offset),
        add_to_open_end(mem(BaseReg,Offset), ThisUse),
        prevent_usage(OppUse),
        memory_compatible(ThisUse).

% Check if all concurrent memory accesses are compatible with each other.
%   arg 1: input:  the memory access resource usage structure to check
%
memory_compatible(Var) :-
        var(Var), !.
memory_compatible([Var|_Rest]) :-
        var(Var), !.
memory_compatible([]) :- !.
memory_compatible([mem(BaseReg,Offset)|Rest]) :-
        memory_compatible(Rest, BaseReg, Offset).

memory_compatible(Var, _, _) :-
        var(Var), !.
memory_compatible([Var|_Rest], _, _) :-
        var(Var), !.
memory_compatible([], _, _) :- !.
memory_compatible([mem(BaseReg2,Offset2)|Rest], BaseReg, Offset) :-
        BaseReg2 == BaseReg,
        CheckOff is Offset \/ 1,
        CheckOff2 is Offset2 \/ 1,
        CheckOff == CheckOff2,
        memory_compatible(Rest, BaseReg, Offset).


%%%%%%%%%%%%%% Routines for determining register/memory usage %%%%%%%%%%%%%%
```

```
% Determine the registers/memory locations read in doing a memory reference;
%  like "src_use", but restricted to displacement-type arithmetic (+ and -).
%   arg 1: input:  "address" expression from a memory reference
%   arg 2: input:  []
%               (name of last found base register in "address", or [] if none)
%   arg 3: output: name of last base register in "address"; [] if none
%   arg 4: input:  +
%               (+ or -, whether next addr. term should be added or subtracted)
%   arg 5: input:  0
%               (current offset value in "address")
%   arg 6: output: total offset value in "address"
%
addr_use(_Tag@Term, BaseReg0, BaseReg, Sign, Offset0, Offset) :- !,
        addr_use(Term, BaseReg0, BaseReg, Sign, Offset0, Offset).
addr_use(Number, BaseReg, BaseReg, Sign, Offset0, Offset) :-
        number(Number), !,
        (Sign == + ->
                Offset is Offset0 + Number ;
                Offset is Offset0 - Number
        ).
addr_use(Atom, BaseReg0, Atom, Sign, Offset, Offset) :-
        atom(Atom), !,
        (BaseReg0 == [] ->
                true ;
                (write('*** WARNING: memory address specified >1 base register; '),
                 write(BaseReg0), write(' is being ignored'), nl
                )
        ),
        (Sign == + ->
                true ;
                (write('*** WARNING: subtracting base register '),
                 write(Atom), write('; I''m adding it instead.'), nl
                )
        ).
addr_use(Op, BaseReg0, BaseReg, Sign, Offset0, Offset) :-
        functor(Op, +, 2), !,
        arg(1, Op, Term1),
        arg(2, Op, Term2),
        addr_use(Term1, BaseReg0, BaseReg1, Sign, Offset0, Offset1),
        addr_use(Term2, BaseReg1, BaseReg, Sign, Offset1, Offset).
addr_use(Op, BaseReg0, BaseReg, Sign, Offset0, Offset) :-
        functor(Op, -, 2), !,
        arg(1, Op, Term1),
        arg(2, Op, Term2),
        addr_use(Term1, BaseReg0, BaseReg1, Sign, Offset0, Offset1),
        (Sign == + ->
                OppSign = - ;
                OppSign = +
        ),
        addr_use(Term2, BaseReg1, BaseReg, OppSign, Offset1, Offset).
addr_use(Unknown, BaseReg, BaseReg, _Sign, Offset, Offset) :-
        write('*** WARNING: unknown memory address form '),
        write(Unknown), write('; I''m ignoring it'), nl.


% Count the number of each type of resource that was used.
%   arg 1: input:  resource usage structure to be counted (can have any arity)
%   arg 2: input:  1
%              (tracks the next member of resource structure to examine)
%   arg 3: output: structure giving usage count for every tracked resource
%
extract_counts(ResUse, Arg, Counts) :-
        arg(Arg, ResUse, List),
        bound_count(List, 0, Count),
        NextArg is Arg + 1,
        (arg(NextArg, ResUse, _Something) ->
               (Counts = (Count,Rest),
                extract_counts(ResUse, NextArg, Rest)
               ) ;
               Counts = Count
        ).


% Count the number of bound-to-non-nil elements in a list.
%   arg 1: input:  the list to be counted
%   arg 2: input:  0
%              (running sum of elements counted so far)
%   arg 3: output: the total number of bound elements
%
bound_count([], Count, Count) :- !.
bound_count([[]|Rest], C0, Count) :- !,        % Nil or unbound
        bound_count(Rest, C0, Count).
bound_count([_|Rest], C0, Count) :-
        C1 is C0 + 1,
        bound_count(Rest, C1, Count).


%%%%%%%%%%%%%%%%%% Routines to deal with open-ended lists %%%%%%%%%%%%%%%%%

% Add an element to an open end list.
%   arg 1: input:  the element to be added
%   arg 2: input:  the open end list to receive the new element
%
add_to_open_end(Element, [Element|_]) :- !.
add_to_open_end(Element, [_|Rest]) :-
    add_to_open_end(Element, Rest).

% Add a *list* of elements to an open end list.
%   arg 1: input:  list of elements to be added
%   arg 2: input:  the open end list to receive the new elements
%
add_list_to_open_end([], _) :- !.
add_list_to_open_end([First|Rest], List) :-
    add_to_open_end(First, List),
    add_list_to_open_end(Rest, List).

% Peel off a given number of elements from head of an open-ended list.
%   arg 1: input:  number of elements to peel off
%   arg 2: input:  open ended list to be peeled
%   arg 3: output: pointer into open ended list after doing the peeling
%
peel_off(0, List, List) :- !.
peel_off(Num, [_|L0], List) :-
        NewNum is Num-1,
        peel_off(NewNum, L0, List).

% Tie up the end of an open-end list, so that it cannot get any more elements.
%   arg 1: input:  the open end list to be finished off
%
tie_up_loose_end([]) :- !.
tie_up_loose_end([_|Rest]) :-
    tie_up_loose_end(Rest).


%%%%%%%%%%%%%%%%%%%%%%%%%% Miscellaneous subroutines %%%%%%%%%%%%%%%%%%%%%%%%%

% The ubiquitous list append/concat routine.
%   arg 3 is the concatenation of arg 1 and arg 2
%
append([], List, List) :- !.
append([First | Rest], List, [First | New]) :-
        append(Rest, List, New).


% Pretty-print a (possibly nested) list, one (indented) element per line.
%   arg 1: input:  the list to be pretty-printed
%
writelst(List) :- writelst(List, 0).

writelst([First | Rest], Num) :-
    !,
    tab(Num), write('['), nl,
    NewNum is Num+1,
    writesub([First | Rest], NewNum).
writelst(NonList, Num) :-
    tab(Num), writeq(NonList).

writesub([], Num) :- !,
    NewNum is Num-1,
    tab(NewNum), write(']').
writesub([Only], Num) :- !,
    writelst(Only, Num), nl,
    writesub([], Num).
writesub([First | Rest], Num) :-
    writelst(First, Num), write(','), nl,
    writesub(Rest, Num).
```

A.4 Microinstruction Simulator

A.4.1 Main Simulator

```
/**********************************************************************
 *                     Microinstruction Simulator                     *
 *                     written by Richard Carlson                     *
 *                       version 2.1    5-18-89                       *
 **********************************************************************/

/*     Copyright 1988,89  The Regents of the University of California      */
/*                         All Rights Reserved                             */


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This simulator executes a program composed of microoperations, showing
% the results of the program and also providing useful information about
% dynamic resource usage, etc.
% The input should come from either the microcode compaction program, or
% from the "pre-simulator" program -- it *must* include resource use info.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Operation:
%  *note* "simwam" or a similar file must be compiled/consulted before
%      running a simulation, to provide model-specific information.
%  "simulate(File)"
%    reads input instruction list from the file "File.compact"
%    writes prompts/results to standard output;
%    writes dynamic resource usage information to the file "File.results"
%  --- or ---
%  "simulate(File, PrintList)"
%    will also print out selected instructions as they are simulated ---
%    "PrintList" is a list of execution cycles that should be shown, and
%    can include single cycle numbers or cycle ranges (of the form "X-Y");
%    "PrintList" can also be the keywords "all" or "none".
%
% Format of the input instruction list:  [same as compactor output format]
%  A "{}" structure, containing resource tracking information, then
%  A list of "instructions", where each instruction is a list of:
%    0 or more operations, followed by one "{}" resource usage structure
%
% Possible microoperations within the instruction list:
%  label(Label)        Creates a destination point for a goto operation
%  goto(DestList)      Changes program flow (conditionally or uncond.).
%                      Each element of the DestList can be either
%                        a simple Label, for an unconditional branch; or
%                        (Source,Label), for a branch only if Source is true;
%                        see the list of legal conditions below.
%  Dest <-- Source     Transfers the contents of Source into Dest.
%                      See the list below for legal Source and Dest values.
%  set([Dest=Source,..])"Free" transfer of Source into Dest; i.e., no
%                        resources are used by this operation.  Used to
%                        initialize constants, etc.
%                      Source can include lists, structures, ^'s to indicate
%                        dereferencing, ^^'s to indicate unbound variables,
%                        and {user} to read the source from the terminal.
%                       "set"s and "show"s are never re-ordered.
% show([Dest,...])      Prints the current value of Dest on the terminal.
%                       Dest can also be {Atom}, where the Atom itself is
%                         printed (instead of trying to look up its value).
%                       "set"s and "show"s are never re-ordered.
% end                   Indicates the end of a program's execution.
%
% Resource tracking information:
% The initial tracking information structure has the form
%   {n, Res1Name, Res2Name, ... , ResnName}
%   where "n" is the number of different resources being tracked; and
%   each "ResName" is a symbolic name describing a tracked resource
% The resource usage structure in each instruction has the form
%   {Res1Value, Res2Value, ... , ResnValue}
%   where each "ResValue" indicates the use of the corresponding resource
%
% Legal "Source" values:
% constant value        specifies a literal value to transfer
% tag@value             specifies a literal tag and value
% atom                  specifies a register (or other fixed location)
% mem(Address)          specifies a memory location -- Address can
%                       specify an offset from any base register
% +,-                   perform addition or subtraction
% /\,\/,~               perform logical and, or, negation
% ==,\==                compare both tag and value      \ Comparison operations;
% ?,\?                  compare only tags                >evaluate to 1 if true,
% =,\=,<,>,=<,>=        compare only values             / or 0 if false.
%
% Legal "Dest" values:
% atom                  specifies a register (or other fixed location)
% mem(Address)          specifies a memory location -- Address can
%                       specify an offset from any base register
%
% Format of the output resource usage list:
% A one-line header giving the number of tracked resources & symbolic names,
% Then one line of resource usage information for each simulated cycle.
% All fields are separated by spaces; this output is *not* Prolog-readable.
%
% Header information:
%      <Num> <space> <Name1> <space> <Name2> <space>...<space> <NameNum>
%   There are exactly "Num" names given on the line; each name is simply
%   passed to the output as it was given in the input header line.
% Usage information:
%      <Use1> <space> <Use2> <space>...<space> <UseNum>
%   Each "Use" gives the (numeric) usage of the corresponding resource
%   during one cycle of simulated program execution.

:- op(  980, xfx, >>>). % (just used for printing out instructions)
:- op(  950, xfx, <--). % destructive assignment
:- op(  700, xfx, \=).  % test for value inequality
:- op(  700, xfx, ?).   % test for type equality
:- op(  700, xfx, \?).  % test for type inequality
:- op(  690, fy, ^).    % "pointer to"
:- op(  400, fy, ~).    % bitwise complement operator
:- op(  300, fx, ^^).   % "unbound variable"
:- op(  250, xfx, @).   % concatenation of type and value fields
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%% Top-level simulation predicates %%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Top-level call to simulate a micro-instruction program.
%   arg 1: input:  filename of program to execute (without .compact extension)
%
simulate(InName) :-
        simulate(InName, none).

% Top-level call to simulate a program and show its execution.
%   arg 1: input:  filename of program to execute (without .compact extension)
%   arg 2: input:  execution cycles that should be traced on standard output
%               "all" to show every execution cycle; or
%               "none" to show no execution cycles; or
%               a list whose elements are...
%                       "X" to show execution cycle number X, and
%                       "X-Y" to show execution cycles from X through Y.
%
simulate(InName, PrintList) :-
        make_in_name(InName, NewInName),
        see(NewInName),
        read(UsageStruct),
        UsageStruct = {UsageInfo},
        read(Program),
        seen,
        makeplist(PrintList, NewPrintList),
        statistics(runtime,[_,_]),
        labelinit(Program, NewProg, LabelTree),
        name(InName, InList),
        convert_name(InList, OutList),
        name(OutName, OutList),
        write('Starting simulation...'), ttyflush,
        tell(OutName),
        writeusageinfo(UsageInfo),
        tosim(NewProg, _+[], 0, [], _FState, FTime, -1/NewPrintList, LabelTree),
        told,
        statistics(runtime,[_,RunTime]),
        nl, write('Simulation complete; total elapsed runtime = '),
        write(RunTime), write('ms'), nl,
        write('Simulation completed after instruction '), write(FTime), nl.


% Convert a base filename into the filename of the input program.
%   arg 1: input:  the base filename (with no extension)
%   arg 2: output: the filename with ".compact" extension
%
make_in_name(InName, NewInName) :-
        name(InName, InList),
        name('.compact', InSuffix),
        append(InList, InSuffix, NewInList),
        name(NewInName, NewInList).
```

```
% Convert an input filename into an output filename.
%   arg 1: input:  the input filename (with or without ".compact" extension)
%   arg 2: output: the output filename (with ".results" extension)
%
convert_name(InSuffix, OutSuffix) :-
        name('.compact', InSuffix), !,
        name('.results', OutSuffix).
convert_name([], OutSuffix) :- !,
        name('.results', OutSuffix).
convert_name([Char | InRest], [Char | OutRest]) :-
        convert_name(InRest, OutRest).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% The actual simulation procedures %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Prepare to simulate an operation --- detect end-of-program (and stop), and
%    end-of-instruction (and tally resource usage info); otherwise,
%    simulate the next instruction by calling "sim".
%   arg 1: input:  the remaining instruction list to be simulated
%   arg 2: input:  the current program state (memory, regs, etc.)
%   arg 3: input:  the current simulation time (in cycles)
%   arg 4: input:  symbolic representation of this instruction's operations
%   arg 5: output: the final simulation state
%   arg 6: output: the final simulation time
%   arg 7: input:  the current list of instructions to be printed out
%   arg 8: input:  tree mapping labels to sections of code
%
                % End-of-program; terminate the recursion
tosim([[]], FState, FTime, _CurrInstr, FState, FTime, _InstrList, _LabelTree) :-
    !.
                % End of instruction; handle usage info then go to next instr.
tosim([[{Info}] | Rest], State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    !,
    NewTime is Time + 1,
    timewarp(State, NewState),
    formatinstr(NewTime, Info, CurrInstr, IList, NewIList),
    tosim(Rest, NewState, NewTime, [], FState, FTime, NewIList, LTree).
tosim([[Instr|Par] | Ser], State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    sim(Instr, [Par | Ser], State, Time, CurrInstr, FState, FTime, IList, LTree).

% Similar to "tosim", except handles a branch in program flow.
%   [args 1,3-9 are same as args 1,2-8 for "tosim"]
%   arg 2: input:  new starting code location after current instruction
%
gotosim([Par | _Ser], NewCode, State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    tosim([Par | NewCode], State, Time, CurrInstr, FState, FTime, IList, LTree).

% Simulate one operation within one instruction, then recur to "tosim".
%   arg 1: input:  the operation to simulate now
%   arg 2: input:  the remaining operations to simulate
%   arg 3: input:  the current program state (memory, regs, etc.)
%   arg 4: input:  the current simulation time (in cycles)
%   arg 5: input:  symbolic representation of this instruction's operations
%   arg 6: output: the final simulation state
%   arg 7: output: the final simulation time
%   arg 8: input:  the current list of instructions to be printed out
%   arg 9: input:  tree mapping labels to sections of code
%
                % Execute an assignment operation
sim((E1<--E2), Rest, State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    !,
    evalexpr(E2, Val2, State, State1),
    evaldestexpr(E1, Val1, State1, State2),
    assign(Val1, Val2, State2, NewState),
    tosim(Rest, NewState, Time, [((E1 <-- E2) >>> (Val1 <-- Val2)) | CurrInstr],
                                FState, FTime, IList, LTree).
                % Execute an assignment of a program location (label)
sim(loadlabel(E1,E2), Rest, State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    !,
    Val2 = label@E2,
    evaldestexpr(E1, Val1, State, State1),
    assign(Val1, Val2, State1, NewState),
    tosim(Rest, NewState, Time, [(loadlabel(E1,E2) >>> (Val1 <-- Val2)) | CurrInstr],
                                FState, FTime, IList, LTree).
                % Conditionally jump
sim(goto([(Cond,Label)|RG]), Rest, State, Time, CurrInstr,
                                FState, FTime, IList, LTree) :-
    !,
    evalexpr(Cond, Val, State, NewState),
    ((Val = _Tag@0) ->
        sim(goto(RG), Rest, NewState, Time, [(goto(Cond, Label) >>> **) | CurrInstr],
                                FState, FTime, IList, LTree) ;
        (followlabel(Label, NewCode, LTree),
         gotosim(Rest, NewCode, State, Time,
                                [(goto(Cond, Label) >>> goto(Label)) | CurrInstr],
                                FState, FTime, IList, LTree)
        )
    ).
                % Jump to a new program location -- indirect
sim(goto([[IndLabel]]), Rest, State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    !,
    evalexpr(IndLabel, Value, State, NewState),
    (Value = label@NewLab ->
        (label(NewLab, NewCode, LTree),
         gotosim(Rest, NewCode, NewState, Time, [goto([IndLabel])|CurrInstr],
                                FState, FTime, IList, LTree)
        ) ;
        (write('*** ERROR in sim:  '), write(IndLabel), write(' = '),
         write(Value), write(' is not a label'), nl, ttyflush,
         fail
        )
    ).
                % Jump to a new program location
sim(goto([Label|_RG]), Rest, State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    !,
    followlabel(Label, NewCode, LTree),
    gotosim(Rest, NewCode, State, Time, [goto(Label) | CurrInstr],
                                FState, FTime, IList, LTree).
                % End the program thread
```

```
sim(end, Rest, State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    !,
    gotosim(Rest, [[]], State, Time, [end | CurrInstr],
                                FState, FTime, IList, LTree).
                % Initialize memory (zero-cycle, memory conflicts ignored)
sim(set(List), Rest, State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    !,
    telling(File),
    tell(user),
    setstate(List, Time, State, NewState),
    tell(File),
    tosim(Rest, NewState, Time, [setstate | CurrInstr],
                                FState, FTime, IList, LTree).
                % Show memory state (zero-cycle, memory conflicts ignored)
sim(show(List), Rest, State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    !,
    telling(File),
    tell(user),
    nl, showstate(List, State, NewState), nl,
    tell(File),
    tosim(Rest, NewState, Time, [showstate | CurrInstr],
                                FState, FTime, IList, LTree).
                % ("label" operations were removed during preprocessing)
                % We don't know how to deal with anything else.
sim(Unk, Rest, State, Time, CurrInstr, FState, FTime, IList, LTree) :-
    write('Error -- unknown instruction (ignored):  '), write(Unk), nl,
    tosim(Rest, State, Time, CurrInstr, FState, FTime, IList, LTree).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% Procedures for evaluating expression values. %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Evaluate a "destination" expression as a memory or register address.
%   arg 1: input:  the address expression to be evaluated
%   arg 2: output: the value after evaluating the address expression --
%       will either be a register ('h') or specific memory location
%       ('mem(Num)' where Num has a *numeric* value)
%   arg 3: input:  the current simulation state
%   arg 4: output: the simulation state after evaluating this address
%
                % An atom refers to a register
evaldestexpr(Atom, Atom, State, State) :-
    atom(Atom),
    !.
                % Make sure memory address is a number
evaldestexpr(mem(E1), mem(Val), State, NewState) :-
    evalexpr(E1, Val, State, NewState).


% Evaluate a "source" expression as a data value.
%   arg 1: input:  the data expression to be evaluated
%   arg 2: output: the value after evaluating the data expression --
%       will always be a primitive value
%   arg 3: input:  the current simulation state
%   arg 4: output: the simulation state after evaluating this expression
%
                            % Constants
evalexpr(Const, NewConst, State, State) :-
    tagdefault(Const, NewConst),
    !.
evalexpr(Type@Const, Type@Const, State, State) :-
    constant(Const),
    !.
                            % Registers or memory locations
evalexpr(Atom, Value, State, NewState) :-
    evaldestexpr(Atom, NewAtom, State, State1),
    lookup(NewAtom, Value, State1, NewState),
    !.
                            % Unary operations
evalexpr((~ E1), T1@Val, State, NewState) :-
    !,
    evalexpr(E1, (T1@D1), State, NewState),
    Val is \(D1).
evalexpr(type(E1), T1, State, NewState) :-
    !,
    evalexpr(E1, (T1@_D1), State, NewState).
evalexpr(value(E1), D1, State, NewState) :-
    !,
    evalexpr(E1, (_T1@D1), State, NewState).
                            % Binary operations
evalexpr((T1 @ E2), Type@Val, State, NewState) :-
    !,
    evaltag(T1, Type, State, State1),
    evalexpr(E2, (_T2@Val), State1, NewState).
evalexpr((E1 /\ E2), T1@Val, State, NewState) :-
    !,
    evalexpr(E1, (T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    Val is D1 /\ D2.
evalexpr((E1 \/ E2), T1@Val, State, NewState) :-
    !,
    evalexpr(E1, (T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    Val is D1 \/ D2.
evalexpr((E1 + E2), T1@Val, State, NewState) :-
    !,
    evalexpr(E1, (T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    Val is D1 + D2.
evalexpr((E1 - E2), T1@Val, State, NewState) :-
    !,
    evalexpr(E1, (T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    Val is D1 - D2.
evalexpr((E1 = E2), none@Val, State, NewState) :-
    !,
    evalexpr(E1, (_T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    ((D1 == D2) -> Val is -1 ; Val is 0).
evalexpr((E1 \= E2), none@Val, State, NewState) :-
```

```
    !,
    evalexpr(E1, (_T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    ((D1 \== D2) -> Val is -1 ; Val is 0).
evalexpr((E1 < E2), none@Val, State, NewState) :-
    !,
    evalexpr(E1, (_T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    ((D1 < D2) -> Val is -1 ; Val is 0).
evalexpr((E1 =< E2), none@Val, State, NewState) :-
    !,
    evalexpr(E1, (_T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    ((D1 =< D2) -> Val is -1 ; Val is 0).
evalexpr((E1 > E2), none@Val, State, NewState) :-
    !,
    evalexpr(E1, (_T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    ((D1 > D2) -> Val is -1 ; Val is 0).
evalexpr((E1 >= E2), none@Val, State, NewState) :-
    !,
    evalexpr(E1, (_T1@D1), State, State1),
    evalexpr(E2, (_T2@D2), State1, NewState),
    ((D1 >= D2) -> Val is -1 ; Val is 0).
evalexpr((E1 == E2), none@Val, State, NewState) :-
    !,
    evalexpr(E1, V1, State, State1),
    evalexpr(E2, V2, State1, NewState),
    ((V1 == V2) -> Val is -1 ; Val is 0).
evalexpr((E1 \== E2), none@Val, State, NewState) :-
    !,
    evalexpr(E1, V1, State, State1),
    evalexpr(E2, V2, State1, NewState),
    ((V1 \== V2) -> Val is -1 ; Val is 0).
evalexpr((E1 ? E2), none@Val, State, NewState) :-
    !,
    evaltag(E1, T1, State, State1),
    evaltag(E2, T2, State1, NewState),
    ((T1 == T2) -> Val is -1 ; Val is 0).
evalexpr((E1 \? E2), none@Val, State, NewState) :-
    !,
    evaltag(E1, T1, State, State1),
    evaltag(E2, T2, State1, NewState),
    ((T1 \== T2) -> Val is -1 ; Val is 0).


% Extract the "tag" part of a data expression
%   arg 1: input:  the data expression to be evaluated -- either a tag
%       alone, or a "type" operation on a source expression
%   arg 2: output: the tag value of the data expression
%   arg 3: input:  the current simulation state
%   arg 4: output: the simulation state after evaluating this expression
%
evaltag(Atom, Atom, State, State) :-
    atom(Atom),
    !.
```

```
evaltag(type(E1), T1, State, NewState) :-
    evalexpr(E1, (T1@_D1), State, NewState).


% Determine if a given value is a "constant"; fail if it is not.
%   arg 1: input:  the value to evaluate -- currently, numbers, the
%       special symbols "nil" and "eos", and structure names of the
%       form "Name/Arity" are considered to be constants
%
constant(Num) :-
    number(Num), !.
constant(nil) :- !.
constant(eos) :- !.
constant((Name/Arity)) :-
    atom(Name),
    number(Arity).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Procedures for handling symbolic name references and assignments. %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Look up the value of a register/memory location.
%   arg 1: input:  the symbolic name of the register/memory to examine
%   arg 2: output: the current value at that location
%   arg 3: input:  the current simulation state
%   arg 4: output: the new simulation state after looking up this value
%
lookup(Name, Value, State+Future, State+Future) :-
        atom(Name), !,
        temp_tree_lookup(Name, Value, State).
lookup(mem(_Tag@Addr), Value, State+Future, State+Future) :-
        BigAddr is Addr + 100000,
        fold_addr(BigAddr, 0, FoldAddr),
        temp_tree_lookup(mem(FoldAddr), Value, State).

% Assign a new value to a register/memory location -- note that this
%    assignment actually must not take effect until the end of this
%    instruction cycle.
%   arg 1: input:  the symbolic name of the register/memory to change
%   arg 2: input:  the new value to store at that location
%   arg 3: input:  the current simulation state
%   arg 4: output: the new simulation state after storing this value
%
assign(Name, Value, State, NewState) :-
        atom(Name), !,
        futureassign([Name | Value], State, NewState).
assign(mem(_Tag@Addr), Value, State, NewState) :-
        BigAddr is Addr + 100000,
        fold_addr(BigAddr, 0, FoldAddr),
        futureassign([mem(FoldAddr) | Value], State, NewState).

% Save a new value assignment in the simulation state structure.
%   arg 1: input:  the new [Name | Value] assignment pair
%   arg 2: input:  the current simulation state
%   arg 3: output: the new simulation state after storing this value
%
futureassign(Item, State+Future, State+[Item | Future]).


% "Reverse" an address number, to un-serialize sequential memory addresses
%    i.e., addresses 1234, 1235, 1236, 2234, 2235 turn into
%                    4321, 5321, 6321, 4322, 5322
%   arg 1: input:  the original address
%   arg 2: input:  the partial reversed address so far (initially, 0)
%   arg 3: output: the completed reversed address
%
fold_addr(0, Fold, Fold) :- !.
fold_addr(Num, Temp, Fold) :-
        NextNum is Num mod 10,
        RestNum is Num // 10,
        NewTemp is (Temp * 10) + NextNum,
        fold_addr(RestNum, NewTemp, Fold).


% Make all of the assignments during the previous instruction permanent.
%   arg 1: input:  the current simulation state
%   arg 2: output: the simulation state, after making all "future"
%       assignments to the "permanent" state
%
timewarp(State+[], State+[]) :- !.
timewarp(State+[[Loc|Val] | Prev], NewState+NewPrev) :-
                % Note ordering of next two clauses doesn't matter, as
                % long as we do only one write per cycle to a location
                % (otherwise, the stores must be done in reverse order,
                % and that prevents tail recursion).
        temp_tree_store(Loc, Val, State, MyState),
        timewarp(MyState+Prev, NewState+NewPrev).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% Procedures for handling label definitions and references. %%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Initialize label tree data structure, finding all labels in the original
%    program, removing them from the code, and inserting a pair mapping
%    the label to the code address into the label tree.
%   arg 1: input:  the original input program
%   arg 2: output: the new program, with labels removed
%   arg 3: output: the completed label tree structure
%
labelinit([], [], _LabelTree) :- !.
labelinit([Instr|Rest], NewCode, LabelTree) :-
    NewCode = [NewInstr|NewRest],
    labelscan(Instr, NewInstr, NewCode, LabelTree),
    labelinit(Rest, NewRest, LabelTree).

% Gather all labels that refer to the current program location.
%   arg 1: input:  pointer into the input program
%   arg 2: output: new pointer, past any label instructions
%   arg 3: input:  pointer to code that current labels point to
%   arg 4: input:  the label tree structure
%
labelscan([label(Label)|Rest], NewInstr, NewCode, LabelTree) :-
        perm_tree_store(Label, NewCode, LabelTree),
        var(NewInstr), !,       % duplicate label will bind NewInstr
        labelscan(Rest, NewInstr, NewCode, LabelTree).
labelscan([label(Label)|_Rest], _, _, _LabelTree) :- !,
        write('*** ERROR in labelscan: duplicate label '), write(Label), nl,
        fail.
                     % New instruction starts with first non-label op
labelscan(NewInstr, NewInstr, _NewCode, _LabelTree).


% Map a given label into its corresponding code.
%   arg 1: input:  the label to look up
%   arg 2: output: the corresponding code
%   arg 3: input:  the label tree structure
%
label(Label, Code, LabelTree) :-
        perm_tree_lookup(Label, Code, LabelTree).

% Same as "label", but follow "goto" instructions to get to "real" code.
%
followlabel(Label, Code, LabelTree) :-
        label(Label, TempCode, LabelTree),
        TempCode = [[goto([Dest])|_]|_],
        atom(Dest), !,
        followlabel(Dest, Code, LabelTree).
followlabel(Label, Code, LabelTree) :-
        label(Label, Code, LabelTree).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% Procedures for printing out the simulation results nicely. %%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Convert special keywords "all" and "none" into valid print lists
%   arg 1: input:  user-specified print list, or "all" or "none"
%   arg 2: output: equivalent print list
%
makeplist([A|B], [A|B]).
makeplist(all, [1-0]).
makeplist(none, [0-1]).

% Print out resource usage, and the instruction if the user wants to.
%   arg 1: input:  the current simulation time (instruction count)
%   arg 2: input:  resource usage information structure
%   arg 3: input:  symbolic representation of what this instruction did
%   arg 4: input:  structure telling which instructions to print out
%   arg 5: output: new structure telling which instructions to print out
%
formatinstr(Time, Info, Instrs, LastSkip/ToDo, LastSkip/NewToDo) :-
    writesequence(Info),
    (donum(Time, ToDo, NewToDo) ->
        (telling(File),
         tell(user),
         rightjust(Time, 5), write(':'),
         subformatinstr(Instrs), nl,
         tell(File)
        ) ;
        NewToDo = ToDo
    ).

% Print out resource usage information.
%   arg 1: input:  the instruction's resource usage structure
%
writeusageinfo(UsageInfo) :-
        countsize(UsageInfo, Length),
        write(Length), write(' '),
        writesequence(UsageInfo).

% Count the number of arguments in a resource usage structure
%   arg 1: input:  resource usage structure, from the input program
%   arg 2: output: the number of arguments (number of resources tracked)
%
countsize(Struct, Length) :-
        countsize(Struct, 0, Length).
countsize((_First,Rest), L0, Length) :- !,
        L1 is L0 + 1,
        countsize(Rest, L1, Length).
countsize(_Only, L0, Length) :-
        Length is L0 + 1.

% Write the elements of a structure out, separated by spaces, ended with nl.
%   arg 1: input:  the resource usage structure
%
writesequence((First,Rest)) :- !,
        write(First), write(' '),
        writesequence(Rest).
writesequence(Only) :-
        write(Only), nl.


%%%%%%%%%%%%%%%%% Symbolic instruction-printing subroutines %%%%%%%%%%%%%%%%%

portray(A>>>B) :-
    write(A), write(' >>> '), write(B).
%    write(A).

% Determine if the current instruction should be printed out.
%   arg 1: input:  the current instruction number
%   arg 2: input:  the current print list
%   arg 3: output: new print list, after taking this instruction into account
%
donum(Num, [Num | ToDo], ToDo) :- !.
donum(Num, [Num-Num | ToDo], ToDo) :- !.
donum(Num, [Num-Bound | ToDo], [NewNum-Bound | ToDo]) :-
    NewNum is Num + 1.

% Print out an operation list, separated by lots of spaces.
%   arg 1: input:  list of all operations completed during this instruction
%
subformatinstr([]) :- !.
subformatinstr([First | Rest]) :-
    subformatinstr(Rest),
    write('    '),
    write(First).


% Print Name in a field Width characters wide, right justified.
%
rightjust(Name, Width) :-
    name(Name, String),
    length(String, Len),
    ToGo is Width - Len,
    tab(ToGo),
    print(Name).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%% Miscellaneous subroutines %%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%% Routines to handle binary search trees %%%%%%%%%%%%%%%%%%
% each node of the tree has the form  node(Name, Value, LeftChild, RightChild)

%%%% The first set of routines deals with "permanent" trees, in which each
%%%%  name is associated with only one value for the duration of the program

% Lookup a value in a tree; print error if given name is not already in tree.
%   arg 1: input:  the name to be looked up
%   arg 2: output: the value assigned to that name
%   arg 3: input:  the permanent tree structure
%
perm_tree_lookup(Name, 0, Var) :-
    var(Var), !,
    write('*** Error: can''t look up value of '), print(Name),
    write(' (using the value 0 instead) ***'), nl.
perm_tree_lookup(Name, Value, node(Name, Value, _Left, _Right)) :- !.
perm_tree_lookup(Name, Value, node(CheckName, _CheckValue, Left, _Right)) :-
    Name @< CheckName, !,
    perm_tree_lookup(Name, Value, Left).
perm_tree_lookup(Name, Value, node(CheckName, _CheckValue, _Left, Right)) :-
    Name @> CheckName, !,
    perm_tree_lookup(Name, Value, Right).

% Lookup or store a value in a tree; create a new node if the given name was
%  not already in the tree.
%   [arguments are the same as for "perm_tree_lookup"]
%
perm_tree_store(Name, Value, node(Name, Value, _Left, _Right)) :- !.
perm_tree_store(Name, Value, node(CheckName, _CheckValue, Left, _Right)) :-
    Name @< CheckName, !,
    perm_tree_store(Name, Value, Left).
perm_tree_store(Name, Value, node(CheckName, _CheckValue, _Left, Right)) :-
```

```
    Name @> CheckName, !,
    perm_tree_store(Name, Value, Right).


%%%% The second set of routines deals with "temporary" trees, in which each
%%%%  name may potentially take on many different values

% Lookup a value in a tree; return 'none@0' if given name is not in tree.
%   arg 1: input:  the name to be looked up
%   arg 2: output: the value assigned to that name
%   arg 3: input:  the temporary tree structure
%
temp_tree_lookup(_Name, none@0, []) :- !.       % Return 'none@0' if unknown
temp_tree_lookup(Name, Value, node(Name, Value, _Left, _Right)) :- !.
temp_tree_lookup(Name, Value, node(CheckName, _CheckValue, Left, _Right)) :-
    Name @< CheckName, !,
    temp_tree_lookup(Name, Value, Left).
temp_tree_lookup(Name, Value, node(CheckName, _CheckValue, _Left, Right)) :-
    Name @> CheckName,
    temp_tree_lookup(Name, Value, Right).

% Store a value in a tree; replace any previously stored value.
%   arg 1: input:  the name to be assigned a value
%   arg 2: output: the value to assign to that name
%   arg 3: input:  previous temporary tree structure
%   arg 4: output: new temporary tree structure, with modified value
%
temp_tree_store(Name, Value, [], node(Name, Value, [], [])) :- !.
temp_tree_store(Name, Value, node(Name, _OldValue, Left, Right),
                        node(Name, Value, Left, Right)) :- !.
temp_tree_store(Name, Value, node(CheckName, CheckValue, Left, Right), NewNode) :-
    Name @< CheckName, !,
    NewNode = node(CheckName, CheckValue, NewLeft, Right),
    temp_tree_store(Name, Value, Left, NewLeft).
temp_tree_store(Name, Value, node(CheckName, CheckValue, Left, Right), NewNode) :-
    Name @> CheckName, !,
    NewNode = node(CheckName, CheckValue, Left, NewRight),
    temp_tree_store(Name, Value, Right, NewRight).


%%%%%%%%%%%%%%%%%%%%%%%%%%% Miscellaneous subroutines %%%%%%%%%%%%%%%%%%%%%%%%%%

% The ubiquitous list append/concat routine.
%   arg 3 is the concatenation of arg 1 and arg 2
%
append([], List, List) :- !.
append([First | Rest], List, [First | New]) :-
        append(Rest, List, New).
```

A.4.2 Auxiliary Model File

```
/*****************************************************************************
 *                         Abstract  Flowchart  Simulator                    *
 *        System-Dependent Procedures for the Warren Abstract Machine        *
 *****************************************************************************/

/*    Copyright 1988,1989  The Regents of the University of California     */
/*                           All Rights Reserved                           */


setstate([], _Time, State, State) :- !.
setstate([(Dest = Value) | Rest], Time, State, StateOut) :-
        evaldestexpr(Dest, NewDest, State, State1),
        setone((NewDest = Value), NewDest, State1, State2, _LastUsed),
        timewarp(State2, NewState),
        setstate(Rest, Time, NewState, StateOut).

setone((Dest = {user}), LastUsed, State, NewState, MyLastUsed) :- !,
        write('Please enter initial value for '), write(Dest), write(': '),
        read(Value), nl,
        setone((Dest = Value), LastUsed, State, NewState, MyLastUsed).
setone((Dest = ^ mem(Addr)), LastUsed, State, NewState, LastUsed) :-
        !,
        evalexpr((const@Addr), NewAddr, State, State1),
        assign(Dest, NewAddr, State1, NewState).
setone((mem(Addr) = ^ Value), mem(LastUsed), State, NewState, MyLastUsed) :-
        !,
        evalexpr((var@LastUsed+1), NewDest, State, State1),
        assign(mem(Addr), NewDest, State1, State2),
        !,
        setone((mem(NewDest) = Value), mem(NewDest), State2, NewState, MyLastUsed).
setone((mem(Addr) = ^^), LastUsed, State, NewState, LastUsed) :-
        !,
        evalexpr((var@Addr), VarAddr, State, State1),
        assign(mem(Addr), VarAddr, State1, NewState).
setone((Dest = []), LastUsed, State, NewState, LastUsed) :-
        !,
        assign(Dest, (const@nil), State, NewState).
setone((mem(Addr) = [First | Rest]), mem(LastUsed), State, NewState, MyLastUsed) :-
        !,
        evalexpr((list@LastUsed+1), NewDest, State, State1),
        assign(mem(Addr), NewDest, State1, State2),
        evalexpr((NewDest+1), NewDest1, State2, State3),
        setone((mem(NewDest) = First), mem(NewDest1), State3, State4, NextLastUsed),
        setone((mem(NewDest1) = Rest), NextLastUsed, State4, NewState, MyLastUsed).
setone((Dest = constant), LastUsed, State, NewState, LastUsed) :-
        !,
        assign(Dest, const@Dest, State, NewState).
setone((Dest = Value), LastUsed, State, NewState, LastUsed) :-
        evalexpr(Value, NewValue, State, State1),
        !,
        assign(Dest, NewValue, State1, NewState).
setone((mem(Addr) = Struct), mem(LastUsed), State, NewState, MyLastUsed) :-
        functor(Struct, Func, Arity),
        !,
        evalexpr((LastUsed \/ 1), OddLastUsed, State, State0),
        evalexpr((struct@OddLastUsed+1), NewDest, State0, State1),
        assign(mem(Addr), NewDest, State1, State2),
        assign(mem(NewDest), const@Func, State2, State3),
        assign(Func, const@Func, State3, State4),
        evalexpr((NewDest+1), NewDest1, State4, State5),
        assign(mem(NewDest1), const@Arity, State5, State6),
        evalexpr((NewDest1+1), NewDest2, State6, State7),
        evalexpr((NewDest1+Arity), NextUsed, State7, State8),
        setstruct(mem(NewDest2), Struct, 1, Arity, mem(NextUsed),
                                 State8, NewState, MyLastUsed).
setone(Unknown, _, State, State, _) :-
        write('*** ERROR in setone: unknown setting '), write(Unknown), nl.

setstruct(mem(_Addr), _Struct, Arg, Arity, mem(LastUsed),
                                 State, State, mem(LastUsed)) :-
        Arg > Arity,
        !.
setstruct(mem(Addr), Struct, Arg, Arity, LastUsed,
                                 State, NewState, MyLastUsed) :-
        Arg =< Arity,
        !,
        arg(Arg, Struct, Value),
        setone((mem(Addr) = Value), LastUsed, State, State1, NextUsed),
        evalexpr((Addr+1), NextAddr, State1, State2),
        NextArg is Arg + 1,
        setstruct(mem(NextAddr), Struct, NextArg, Arity, NextUsed,
                                 State2, NewState, MyLastUsed).
setstruct(Where, What, Arg, Arity, _, State, State, _) :-
        write('*** ERROR in setstruct: setting '),
        write(What), write(', arg '), write(Arg), write(' of '), write(Arity),
        write(', at location '), write(Where), nl.


showstate([], State, State) :- !.
showstate([{String} | Rest], State, NewState) :-
        !,
        write(String), nl,
        showstate(Rest, State, NewState).
showstate([Src | Rest], State, NewState) :-
        evaldestexpr(Src, NewSrc, State, State1),
        lookup(NewSrc, NewVal, State1, State2),
        showone(NewVal, NewSrc, State2, State3, Value),
        write(Src), write(' = '), write(Value), nl,
        showstate(Rest, State3, NewState).

showone(var@MyAddr, mem(Addr), State, NewState, ^^MyAddr) :-
        evalexpr((var@Addr), var@MyAddr, State, NewState),
        !.
showone(var@Addr, _Dest, State, NewState, (^ Value)) :-
        !,
        lookup(mem(var@Addr), NewDest, State, State1),
        showone(NewDest, mem(var@Addr), State1, NewState, Value).
showone(const@nil, _Dest, State, State, []) :-
        !.
showone(list@Addr, _Dest, State, NewState, [First | Rest]) :-
```

```
        !,
        lookup(mem(list@Addr), Head, State, State1),
        showone(Head, mem(list@Addr), State1, State2, First),
        evalexpr((Addr+1), NewAddr, State2, State3),
        lookup(mem(NewAddr), Tail, State3, State4),
        showone(Tail, mem(NewAddr), State4, NewState, Rest).
showone(const@Value, _Dest, State, State, Value) :-
        !.
showone(none@Value, _Dest, State, State, {Value}) :-
        !.
showone(struct@Addr, _Dest, State, NewState, Struct) :-
        lookup(mem(struct@Addr), (const@(Func/Arity)), State, State1),
        !,
        evalexpr((Addr+1), NewAddr, State1, State2),
        showstruct(mem(NewAddr), 1, Arity, State2, NewState, Args),
        Struct =.. [Func | Args].
showone(struct@Addr, _Dest, State, NewState, Struct) :-
        lookup(mem(struct@Addr), (const@Func), State, State1),
        !,
        evalexpr((Addr+1), NewAddr, State1, State2),
        lookup(mem(NewAddr), (const@Arity), State2, State3),
        evalexpr((NewAddr+1), NewAddr2, State3, State4),
        showstruct(mem(NewAddr2), 1, Arity, State4, NewState, Args),
        Struct =.. [Func | Args].
showone(Unknown, _Dest, State, State, {Unknown}).

showstruct(mem(_Addr), Arg, Arity, State, State, []) :-
        Arg > Arity,
        !.
showstruct(mem(Addr), Arg, Arity, State, NewState, [First | Rest]) :-
        Arg =< Arity,
        !,
        lookup(mem(Addr), Value, State, State1),
        showone(Value, mem(Addr), State1, State2, First),
        evalexpr((Addr+1), NextAddr, State2, State3),
        NextArg is Arg + 1,
        showstruct(mem(NextAddr), NextArg, Arity, State3, NewState, Rest).
showstruct(Where, Arg, Arity, State, State, _) :-
        write('*** ERROR in showstruct: showing arg '),
        write(Arg), write(' of '), write(Arity),
        write(', at location '), write(Where), nl.


tagdefault(Num, const@Num) :-
        number(Num), !.
tagdefault(nil, const@nil) :- !.
tagdefault(eos, const@eos).
```

A.5 Histogram Generator

```
/***************************************************************************
 *                          Histogram Generator                            *
 *                       written by Richard Carlson                        *
 *                         version 1.0    5-18-89                          *
 ***************************************************************************/

/*      Copyright 1988,89  The Regents of the University of California      */
/*                           All Rights Reserved                            */


#include <stdio.h>

#define PRINTWIDTH 80

#define MAXCOUNT 100
#define MAXNAME 60

#define MAXTRACK 50

char names[MAXTRACK][MAXNAME];
int counts[MAXTRACK][MAXCOUNT];         /* Assumed initialized to 0 ! */
int highcount[MAXTRACK];                /* Assumed initialized to 0 ! */
int maxoccur[MAXTRACK];                 /* Assumed initialized to 0 ! */
int totaloccur[MAXTRACK];               /* Assumed initialized to 0 ! */

main()
{
        int tracked;
        int tracelen;
        int i;

                        /* Get number of and names of tracked resources */
        scanf("%d ", &tracked);
        for (i = 0; i < tracked; i++)
                scanf("%s ", names[i]);

        tracelen = readdata(tracked);
        histprint(tracelen, tracked);
}

                        /* Read resource data and update array values */
                        /* Returns total number of samples (cycles) found */
readdata(tracked)
        int tracked;
{
        int tracelen = 0;
        int temp;
        int i;

        while (scanf("%d ", &temp) > 0) {
                tracelen++;
                counts[0][temp]++;
                totaloccur[0] += temp;
                if (temp > highcount[0])
```

```
                        highcount[0] = temp;
                if (counts[0][temp] > maxoccur[0])
                        maxoccur[0] = counts[0][temp];
                for (i = 1; i < tracked; i++) {
                        scanf("%d ", &temp);
                        counts[i][temp]++;
                        totaloccur[i] += temp;
                        if (temp > highcount[i])
                                highcount[i] = temp;
                        if (counts[i][temp] > maxoccur[i])
                                maxoccur[i] = counts[i][temp];
                }
        }
        return(tracelen);
}

                        /* Turn the array data into nice histograms */
histprint(tracelen, tracked)
        int tracelen, tracked;
{
        int i, j, k;
        int scalefactor;

        printf("##### TOTAL TRACE LENGTH = %d #####\n\n", tracelen);

        for (i = 0; i < tracked; i++) {
                printf(":::::  %s   ---   total used=%d  :::::\n\n",
                                names[i], totaloccur[i]);
                scalefactor = (maxoccur[i]+PRINTWIDTH-6) / (PRINTWIDTH-5);
                if (scalefactor == 0)
                        scalefactor = 1;
                for (j = highcount[i]; j >= 0; j--) {
                        printf("%4d|", j);
                        for (k = (int)((counts[i][j]+scalefactor-1)/scalefactor);
                                        k >= 1; k--)
                                printf("*");
                        printf("\n");
                }
                printf("    +");
                for (k = 6; k <= PRINTWIDTH; k++)
                        printf("-");
                printf("\n%*d^\n\n", PRINTWIDTH-1, (PRINTWIDTH-5)*scalefactor);
        }
}
```

A.6 Hand-Coded Benchmark Programs

A.6.1 concat — General List Concatenation

```
% Full (non-deterministic) concat benchmark.
%
% Equivalent Prolog source code:
%  concat([], L, L).
%  concat([A|X], L, [A|Y]) :- concat(X, L, Y).
%
% Benchmarks used by this project are started with the calls:
%       A:      concat( [1,2,...,30], [31,32], _ )
%       B:      concat( [1,2,...,30], _, [1,2,...,32] )
%       C:      concat( _, [31,32], [1,2,...,32] )
%       D:      concat( _, _, [1,2,...,32] )
%
% Uses registers:  a1, a2, a3, h, s, fp, tr, ul, u1, u2, uret


[
 [      set([mem(99)=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
                      20,21,22,23,24,25,26,27,28,29,30], mem(179)=[31,32],
            mem(199)=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
                      20,21,22,23,24,25,26,27,28,29,30,31,32],
            h= ^mem(1000), s= ^mem(2000), tr= ^mem(10000),
            ul= ^mem(20000)]) ] ,
 [      mem(tr) <-- const@0 ,           % Create "dummy space" on trail
        mem(tr+1) <-- const@0 ,
        tr <-- tr+2 ] ,
 [      mem(s) <-- const@0 ,
        loadlabel(mem(s+1), nosol) ] ,  % Create top-level choice point
 [      mem(s+2) <-- h ,
        mem(s+3) <-- tr ,
        cp <-- s+2 ] ,
 [      mem(s+5) <-- const@0 ] ,
 [      mem(s+6) <-- const@0 ,
        loadlabel(mem(s+7), fail) ,
        fp <-- s+6 ,
        s <-- s+8 ,
        goto([main]) ] ,

 [      label(main) ,
        a1 <-- var@h ,
        a2 <-- var@h+1 ,
        a3 <-- var@h+2 ,
        show([{'preset lists are "list@100", "list@180", and "list@200";'},
              {'or "^^" represents an unbound variable'}]) ,
        set([mem(h)={user}, mem(h+1)={user}, mem(h+2)={user}]) ,
        temp1 <-- h ,
        h <-- h+4 ] ,
 [      mem(s) <-- temp1 ,
        mem(s+1) <-- cp ] ,            % Create a stack frame
 [      mem(s+2) <-- fp ,
        loadlabel(mem(s+3), mainr) ,
        fp <-- s+2 ,
        s <-- s+4 ,
      goto([concat]) ] ,

[     label(mainr) ,
      temp2 <-- fp+2 ] ,
[     oldh <-- mem(temp2) ] ,
[     show([mem(oldh), mem(oldh+1), mem(oldh+2)]) ] ,
[     newpc <-- mem(fp+1) ,
      fp <-- mem(fp) ] ,
[     goto([[newpc]]) ] ,

[     label(nosol) ,
      show([{'no (more) solutions'}]) ] ,
[     end ] ,

[     label(concat) ,
      goto([((type(a1) ? list), cclist),        % Test type of a1
            ((type(a1) ? const), ccnil),
            ((type(a1) ? struct), fail),
            (ccvar0)]) ] ,
[     label(ccvar0) ,
      label(decat) ,
       tmpa <-- a1 ,
       a1 <-- mem(a1) ] ,     % Dereference a1
[     goto([((a1==tmpa), ccvar),      % a1 is unbound
            (decata)]) ] ,
[     label(decata) ,
      goto([((type(a1) ? list), cclist),
            ((type(a1) ? const), ccnil),
            ((type(a1) ? struct), fail),
            (decat)]) ] ,

[     label(ccvar) ,
      mem(s) <-- fp ,         % a1 is an unbound variable
      loadlabel(mem(s+1), ccvar2) ,             % Create a choice point
      s <-- s+2 ,
      temp3 <-- tr-1 ,
      goto([((tr /\ 1), trodd),
            (treven)]) ] ,
[     label(trodd) ,
      temp3a <-- mem(temp3) ] ,
[     mem(tr) <-- temp3a ,
      tr <-- tr+1 ,
      goto([treven]) ] ,
[     label(treven) ,
      mem(s) <-- h ,
      mem(s+1) <-- tr ,
      cp <-- s ,
      s <-- s+2 ] ,
[     mem(a1) <-- const@nil ] ,                 % First , bind a1 to nil
[     mem(tr) <-- a1 ,
      tr <-- tr+1 ,
      goto([ccnil]) ] ,
[     label(ccnil) ,                  % Entry point if a1 was already nil
      u1 <-- a2 ,                     % Try to unify a2 and a3
      u2 <-- a3 ,
      loadlabel(mem(ul), ccvar1) ,
      goto([[(a2 == a3), ccvar1),
           (unify)]) ] ,
[     label(ccvar1) ,
      newpc <-- mem(fp+1) ,
      fp <-- mem(fp) ] ,
[     goto([[newpc]]) ] ,          % Succeed

[     label(ccvar2) ,              % Alternatively, match a1 & a3 heads
      tempfp <-- fp-1 ] ,
[     cp <-- mem(tempfp) ,         % Throw away current choice point
      goto([[(type(a3) ? list), ccvlist),     % Test type of a3
           ((type(a3) ? const), fail),
           ((type(a3) ? struct), fail),
           (devar)]) ] ,
[     label(devar) ,
       tmpa <-- a3 ,
       a3 <-- mem(a3) ] ,      % Dereference a3
[     goto([[(a3==tmpa), unbounda3), % a3 is unbound
           (devara)]) ] ,
[     label(devara) ,
      goto([[(type(a3) ? list), ccvlist),
           ((type(a3) ? const), fail),
           ((type(a3) ? struct), fail),
           (devar)]) ] ,

[     label(unbounda3) ,
      mem(a3) <-- list@h ] ,       % a3 was unbound; make it a list
[     mem(tr) <-- a3 ,
      a3 <-- list@h ,
      tr <-- tr+1 ,
      temph1 <-- var@h+1 ] ,
[     mem(h) <-- var@h ,
      mem(h+1) <-- temph1 ,
      h <-- h+2 ,
      goto([ccvlist]) ] ,
[     label(ccvlist) ,
      mem(a1) <-- list@h ] ,       % Make a1 into a list, too
[     mem(tr) <-- a1 ,
      tr <-- tr+1 ] ,
[     heada3 <-- mem(a3) ,
      taila3 <-- mem(a3+1) ,
      temph1 <-- var@h+1 ] ,
[     mem(h) <-- heada3 ,
      mem(h+1) <-- temph1 ,
      a1 <-- var@h+1 ,
      h <-- h+2 ,
      a3 <-- taila3 ,
      goto([concat]) ] ,           % Recur to handle the rest

[     label(cclist) ,              % a1 is a list
      goto([[(type(a3) ? list), ccllist),     % Test type of a3
           ((type(a3) ? const), fail),
           ((type(a3) ? struct), fail),
           (delist)]) ] ,
[     label(delist) ,
       tmpa <-- a3 ,
          a3 <-- mem(a3) ] ,      % Dereference a3
[        goto([((a3==tmpa), unb0a3),     % a3 is unbound
               (delista)]) ] ,
[        label(delista) ,
         goto([((type(a3) ? list), ccllist),
               ((type(a3) ? const), fail),
               ((type(a3) ? struct), fail),
               (delist)]) ] ,

[        label(unb0a3) ,
         mem(a3) <-- list@h ] ,          % a3 was unbound; make it a list
[        mem(tr) <-- a3 ,
         tr <-- tr+1 ,
         goto([ccloop]) ] ,

[        label(deccl) ,
          tmpa <-- a1 ,
          a1 <-- mem(a1) ] ,      % Dereference a1
[        goto([((a1==tmpa), ccvar),      % a1 is unbound
               (deccla)]) ] ,
[        label(deccla) ,
         goto([((type(a1) ? list), ccloop),
               ((type(a1) ? const), cclend),
               ((type(a1) ? struct), fail),
               (deccl)]) ] ,
[        label(ccloop) ,                 % Inner loop
         temp5 <-- mem(a1) ,
         a1 <-- mem(a1+1) ,
         temp6 <-- list@(h+2) ] ,
[        mem(h) <-- temp5 ,
         mem(h+1) <-- temp6 ,
         temp7 <-- h+1 ,
         h <-- h+2 ,
         goto([((type(a1) ? list), ccloop),
               ((type(a1) ? const), cclend),
               ((type(a1) ? struct), fail),
               (deccl)]) ] ,
[        label(cclend) ,
         mem(temp7) <-- a2 ,             % Tie up the loose end of a3
         goto([((a1 = nil), a1nil),
               (fail)]) ] ,
[        label(a1nil) ,
         newpc <-- mem(fp+1) ,
         fp <-- mem(fp) ] ,
[        goto([[newpc]]) ] ,             % Succeed

[        label(ccllist) ,
         u1 <-- mem(a1) ,                % Try to unify head of a1 and a3
         temp11 <-- mem(a1+1) ] ,
[        u2 <-- mem(a3) ,
         temp13 <-- mem(a3+1) ] ,
[        loadlabel(mem(u1), cclvar1) ,
         goto([((u1 == u2), cclvar1),
               (unify)]) ] ,
[        label(cclvar1) ,
         a1 <-- temp11 ,
       a3 <-- temp13 ,
       goto([concat]) ] ,

[      label(unify) ,
       uret <-- mem(ul) ,
       goto([(type(u1) ? const, unic1),
             (type(u1) ? list, unil1),
             (type(u1) ? struct, unis1),
             (univ1)]) ] ,

[      label(univ1) ,
        tmpu <-- u1 ,
        u1 <-- mem(u1) ] ,
[      goto([((u1==tmpu), u1varnottmpu),       % u1 is unbound
             (univ1a)]) ] ,
[      label(univ1a) ,
       goto([((type(u1) ? const), unic1),
             ((type(u1) ? list), unil1),
             ((type(u1) ? struct), unis1),
             (univ1)]) ] ,

[      label(u1varnottmpu) ,
       goto([((u1 == u2), uuret),
             (u1notequ2)]) ] ,
[      label(u1notequ2) ,
       goto([((type(u2) \? var), univ1x2),
             (univ1v2)]) ] ,
[      label(univ1v2) ,
        tmpu <-- u2 ,
        u2 <-- mem(u2) ] ,
[      goto([((u2==tmpu), univ1unb2),
             (univ1v2a)]) ] ,
[      label(univ1v2a) ,
       goto([((type(u2) ? var), univ1v2),
             (univ1x2)]) ] ,
[      label(univ1unb2) ,
       goto([((u1=u2), uuret), % done if same variables
             (univ1nev2)]) ] ,
[      label(univ1nev2) ,
       goto([((u1 > u2), univ1x2),
             (unix1v2)]) ] ,   % u1 < u2

[      label(univ1x2) ,
       mem(u1) <-- u2 ] ,
[      mem(tr) <-- u1 ,
       tr <-- tr+1 ,
       goto([[uret]]) ] ,

[      label(unix1v2) ,
       mem(u2) <-- u1 ] ,
[      mem(tr) <-- u2 ,
       tr <-- tr+1 ,
       goto([[uret]]) ] ,

[      label(unic1) ,
       goto([(type(u2) ? const, unic1c2),
             (type(u2) ? list, fail),
             (type(u2) ? struct, fail),
             (unic1v2)]) ] ,
[      label(unic1v2) ,
        tmpu <-- u2 ,
        u2 <-- mem(u2) ] ,
[      goto([((u2==tmpu), unix1v2),
             (unic1v2a)]) ] ,
[      label(unic1v2a) ,
       goto([((type(u2) ? const), unic1c2),
             ((type(u2) ? struct), fail),
             ((type(u2) ? list), fail),
             (unic1v2)]) ] ,

[      label(unic1c2) ,
       goto([((u1 == u2), uuret),
             (fail)]) ] ,

[      label(unis1) ,
       goto([((type(u2) ? struct), unis1s2),
             ((type(u2) ? const), fail),
             ((type(u2) ? list), fail),
             (unis1v2)]) ] ,
[      label(unis1v2) ,
        tmpu <-- u2 ,
        u2 <-- mem(u2) ] ,
[      goto([((u2==tmpu), unix1v2),
             (unis1v2a)]) ] ,
[      label(unis1v2a) ,
       goto([((type(u2) ? struct), unis1s2),
             ((type(u2) ? const), fail),
             ((type(u2) ? list), fail),
             (unis1v2)]) ] ,

[      label(unis1s2) ,
       goto([((u1 == u2), uuret),
             (fail)]) ] ,      % actually, recursively unify structures...

[      label(unil1) ,
       goto([((type(u2) ? list), unil1l2),
             ((type(u2) ? const), fail),
             ((type(u2) ? struct), fail),
             (unil1v2)]) ] ,
[      label(unil1v2) ,
        tmpu <-- u2 ,
        u2 <-- mem(u2) ] ,
[      goto([((u2==tmpu), unix1v2),
             (unil1v2a)]) ] ,
[      label(unil1v2a) ,
       goto([((type(u2) ? list), unil1l2),
             ((type(u2) ? const), fail),
             ((type(u2) ? struct), fail),
             (unil1v2)]) ] ,

[      label(unil1l2) ,
       goto([((u1 == u2), uuret),
                (unil112a)]) ] ,
    [     label(unil112a) ,
          head1 <-- mem(u1) ,
          tail1 <-- mem(u1+1) ] ,
    [     head2 <-- mem(u2) ,
          tail2 <-- mem(u2+1) ] ,
    [     mem(ul+2) <-- tail1 ,
          mem(ul+3) <-- tail2 ,
          ul <-- ul+4 ,
          goto([((head1 == head2), uniltail),
                (unil112b)]) ] ,
    [     label(unil112b) ,
          loadlabel(mem(ul), uniltail) ,
          u1 <-- head1 ,
          u2 <-- head2 ,
          goto([unify]) ] ,
    [     label(uniltail) ,
          u1 <-- mem(ul-2) ,
          u2 <-- mem(ul-1) ,
          ul <-- ul-4 ,
          goto([unify]) ] ,

    [     label(uuret) ,
          goto([[uret]]) ] ,

    [     label(fail) ,
          s <-- cp ,
          h <-- mem(cp) ,
          newtr <-- mem(cp+1) ,
          temp20 <-- cp-2 ] ,
    [     fp <-- mem(temp20) ,
          newfa <-- mem(temp20+1) ,
          goto([((tr /\ 1), failodd),
                (failloop)]) ] ,
    [     label(failloop) ,
          temp21 <-- tr-2 ] ,
    [     tr1 <-- mem(temp21+1) ,
          tr2 <-- mem(temp21) ,
          goto([((tr = newtr), newfail),
                (failcont)]) ] ,
    [     label(failcont) ,
          mem(tr1) <-- var@tr1 ] ,
    [     mem(tr2) <-- var@tr2 ,
          tr <-- tr-2 ,
          goto([failloop]) ] ,
    [     label(failodd) ,
          tr <-- tr-1 ] ,
    [     tr1 <-- mem(tr) ] ,
    [     mem(tr1) <-- var@tr1 ,
          goto([failloop]) ] ,
    [     label(newfail) ,
          goto([[newfa]]) ] ,
    [     end ]
   ].
```

A.6.2 naive reverse — Naive List Reversal

```
% Determinate naive reverse benchmark.
%
% Equivalent Prolog source code:
%  nreverse([A|X], Z) :- !, nreverse(X, Y), concat(Y, [A], Z).
%  nreverse([], []).
%
%  concat([], L, L) :- !.
%  concat([A|X], L, [A|Y]) :- concat(X, L, Y).


[
 [      set([mem(99)=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
                      20,21,22,23,24,25,26,27,28,29,30],
             a1=list@100, a2= ^mem(180), mem(180)= ^^,
             h= ^mem(1000), e= ^mem(10000), s= ^mem(20000)]) ] ,
 [      loadlabel(mem(s), mainret) ,
        s <-- s+1 ,
        goto([nreverse]) ] ,
 [      label(mainret) ,
        show([a3]) ,
        end ] ,

 [      label(nreverse) ,
        goto([(type(a1)?list, nrev1),
              (type(a1)?const, nrdone),
              (derefail)]) ] ,
 [      label(derefail) ,
        tmpa <-- a1 ,
        a1 <-- mem(a1) ] ,
 [      goto([((a1==tmpa), nrev1),
              (nrev0a)]) ] ,
 [      label(nrev0a) ,
        goto([((type(a1)?var), derefail),
              (nrev1)]) ] ,
 [      label(nrev1) ,
        x <-- mem(a1) ,
        l0 <-- mem(a1+1) ] ,
 [      mem(e) <-- x ,
        mem(e+1) <-- a2 ,
        e <-- e+2 ] ,
 [      a1 <-- l0 ,
        a2 <-- var@h ,
        mem(h) <-- var@h ,
        h <-- h+2 ] ,
 [      loadlabel(mem(s), ret1) ,
        s <-- s+1 ,
        goto([nreverse]) ] ,
 [      label(ret1) ,
        a1 <-- a2 ,
        a2 <-- list@h ,
        a3 <-- mem(e-1) ,
        x <-- mem(e-2) ,
        e <-- e-2 ] ,
 [      mem(h) <-- x ,
        mem(h+1) <-- const@nil ,
        h <-- h+2 ] ,
[       loadlabel(mem(s), ret2) ,
        s <-- s+1 ,
        goto([concat]) ] ,
[       label(ret2) ,
        a2 <-- a3 ,
        retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
[       goto([[retaddr]]) ] ,

[       label(nrdone) ,
        label(derefa21) ,
        tmpa <-- a2 ,
        a2 <-- mem(a2) ] ,
[       goto([((a2==tmpa), noderefa21),
              (nrdonea)]) ] ,
[       label(nrdonea) ,
        goto([((type(a2)?var), derefa21),
              (noderefa21)]) ] ,
[       label(noderefa21) ,
        mem(a2) <-- a1 ,
        t <-- const@1 ] ,
[       retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
[       goto([[retaddr]]) ] ,

[       label(nrfail) ,
        t <-- const@0 ,
        retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
[       goto([[retaddr]]) ] ,

[       label(concat) ,
        label(derefa31) ,
        tmpa <-- a3 ,
        a3 <-- mem(a3) ] ,
[       goto([((a3==tmpa), noderefa31),
              (derefa31a)]) ] ,
[       label(derefa31a) ,
        goto([((type(a3)?var), derefa31),
              (noderefa31)]) ] ,
[       label(noderefa31) ,
        mem(a3) <-- list@h ,
        goto([((type(a1)?list), loop),
              (ccderef)]) ] ,
[       label(ccderef) ,
        label(derefa12) ,
        tmpa <-- a1 ,
        a1 <-- mem(a1) ] ,
[       goto([((a1==tmpa), ccfail),
              (derefa12a)]) ] ,
[       label(derefa12a) ,
        goto([((type(a1)?list), loop),
              ((type(a1)?struct), ccfail),
              ((type(a1)?const), ccdone),
              (derefa12)]) ] ,
 [      label(loop) ,
        x <-- mem(a1) ,
        a1 <-- mem(a1+1) ,
        newh <-- list@(h+2) ] ,
 [      mem(h) <-- x ,
        mem(h+1) <-- newh ,
        h <-- h+2 ,
        goto([((type(a1)?list), loop),
              ((type(a1)?var), ccderef),
              (ccendloop)]) ] ,
 [      label(ccendloop) ,
        mem(h-1) <-- a2 ,
        t <-- const@1 ] ,
 [      retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
 [      goto([[retaddr]]) ] ,
 [      label(ccdone) ,
        mem(a3) <-- a2 ,
        t <-- const@1 ] ,
 [      retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
 [      goto([[retaddr]]) ] ,
 [      label(ccfail) ,
        t <-- const@0 ,
        retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
 [      goto([[retaddr]]) ]
].
```

A.6.3 diff — Symbolic Differentiation

```
% Modified determinate symbolic differentiation benchmark.
%
% Equivalent Prolog source code:
%
%  d(U+V,X,DU+DV) :- !, d(U,X,DU), d(V,X,DV).
%  d(U-V,X,DU-DV) :- !, d(U,X,DU), d(V,X,DV).
%  d(U*V,X,DU*V+U*DV) :- !, d(U,X,DU), d(V,X,DV).
%  d(U/V,X,(DU*V-U*DV)/(^(V,2))) :- !, d(U,X,DU), d(V,X,DV).
%  d(^(U,N),X,DU*N*(^(U,N1))) :- !, integer(N), N1 is N - 1, d(U,X,DU).
%  d(-U,X,-DU) :- !, d(U,X,DU).
%  d(exp(U),X,exp(U)*DU) :- !, d(U,X,DU).
%  d(log(U),X,DU/U) :- !, d(U,X,DU).
%  d(X,X,1) :- !.
%  d(C,X,0).
%
% Benchmark used by this project is equivalent to the sequence of calls:
%
%  d( ((((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x, x, _ )
%  d( ((((((((x/x)/x)/x)/x)/x)/x)/x)/x)/x, x, _ )
%  d( log(log(log(log(log(log(log(log(log(log(x)))))))))), x, _ )
%  d( (x+1)*((^(x,2)+2)*(^(x,3)+3)), x, _ )
%
% The 'diff' code written for this benchmark assumes:
%  first argument (a1) is an input term
%  second argument (a2) is a ground term
%  third argument (a3) is an output term
%  everything is deterministic, so no choice points are created
%  saved environments contain only heap pointer & variables used after call
%  the procedure reports its success by setting or clearing a "t" flag


[
                           % Initialize memory structures, etc.
 [      set(['x'=constant, '++' =constant, '--' =constant, '*' =constant,
             '/' =constant, '^' =constant, exp=constant, log=constant,
              mem(99)= ((((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x,
             mem(199)= ((((((((x/x)/x)/x)/x)/x)/x)/x)/x)/x,
             mem(299)= log(log(log(log(log(log(log(log(log(log(x)))))))))),
             mem(399)= '++'(x,1)*('++'(^(x,2),2)*('++'(^(x,3),3))),
             h= ^mem(1000), e= ^mem(10000), s= ^mem(20000)]) ] ,

                %%%%%%%%% main program %%%%%%%%%
                           % 'times10(I1), d(I1,x,D1)'
 [      a1 <-- struct@100 ,
        a2 <-- const@x ,
        loadlabel(mem(s), ret1) ,
        s <-- s+1 ,
        goto([diff]) ] ,
 [      label(ret1) ,
        show([t, a3]) ] ,
                           % 'divide10(I2), d(I2,x,D2)'
 [      a1 <-- struct@200 ,
        a2 <-- const@x ,
        loadlabel(mem(s), ret2) ,
       s <-- s+1 ,
       goto([diff]) ] ,
[      label(ret2) ,
       show([t, a3]) ] ,
                       % 'log10(I3), d(I3,x,D3)'
[      a1 <-- struct@300 ,
       a2 <-- const@x ,
       loadlabel(mem(s), ret3) ,
       s <-- s+1 ,
       goto([diff]) ] ,
[      label(ret3) ,
       show([t, a3]) ] ,
                       % 'ops8(I4), d(I4,x,D4)'
[      a1 <-- struct@400 ,
       a2 <-- const@x ,
       loadlabel(mem(s), ret4) ,
       s <-- s+1 ,
       goto([diff]) ] ,
[      label(ret4) ,
       show([t, a3]) ,
       end ] ,

               %%%%%%%%% diff subroutine %%%%%%%%%
                       % Dereference first argument, if necessary
[      label(dderef) ,
       tmpa <-- a1 ,
       a1 <-- mem(a1) ] ,
[      goto([((a1==tmpa), dfail),      % Fail if it is unbound
             (dderefa)]) ] ,
[      label(dderefa) ,
       goto([((type(a1)?var), dderef),
             (diff)]) ] ,
                       % Test the type of the first argument
[      label(diff) ,
       goto([(type(a1)?var, dderef),
             (type(a1)?const, dconst),
             (type(a1)?struct, dstruct),
             (dfail)]) ] ,
[      label(dfail) ,                  % Fail if it's a list
       t <-- const@0 ,
       retaddr <-- mem(s-1) ,
       s <-- s-1 ] ,
[      goto([[retaddr]]) ] ,

                       % Input expression is a constant
[      label(dconst) ,
       goto([((a1==a2), dsame),
            (dnotsame)]) ] ,
[      label(dnotsame) ,
       a3 <-- const@0 ,                % 'd(X, X, 1)'
       t <-- const@1 ,
       retaddr <-- mem(s-1) ,
       s <-- s-1 ] ,
[      goto([[retaddr]]) ] ,
[      label(dsame) ,
       a3 <-- const@1 ,                % 'd(C, X, 0)'
       t <-- const@1 ,
       retaddr <-- mem(s-1) ,
       s <-- s-1 ] ,
[      goto([[retaddr]]) ] ,

                       % Input expression is a structure
[      label(dstruct) ,
       func <-- mem(a1) ,
       arity <-- mem(a1+1) ] ,
                                       % Branch to appropriate handler
[      goto([((arity==const@2), dar2),
             (dar0)]) ] ,
[      label(dar0) ,
       goto([((arity==const@1), dar1),
             (dfail)]) ] ,             % Fail if unknown structure
[      label(dar2) ,
       goto([((func==const@(++)), dpl2),
             (dar2a)]) ] ,
[      label(dar2a) ,
       goto([((func==const@(--)), dmi2),
             (dar2b)]) ] ,
[      label(dar2b) ,
       goto([((func==const@(*)), dti2),
             (dar2c)]) ] ,
[      label(dar2c) ,
       goto([((func==const@(/)), ddi2),
             (dar2d)]) ] ,
[      label(dar2d) ,
       goto([((func==const@(^)), dpo2),
             (dfail)]) ] ,
[      label(dar1) ,
       goto([((func==const@(--)), dmi1),
             (dar1a)]) ] ,
[      label(dar1a) ,
       goto([((func==const@(exp)), dex1),
             (dar1b)]) ] ,
[      label(dar1b) ,
       goto([((func==const@(log)), dlo1),
             (dfail)]) ] ,

                       % 'd(U++V,X,DU++DV) :- d(U,X,DU), d(V,X,DV).'
[      label(dpl2) ,
       u <-- mem(a1+2) ,               % Extract "u","v" from input structure
       v <-- mem(a1+3) ] ,
[      mem(h) <-- const@(++) ,         % Create output structure "++"/2
       mem(h+1) <-- const@2 ,
       h <-- h+4 ] ,
[      mem(e) <-- v ,                  % Save "v" and "x" on stack
       mem(e+1) <-- a2 ,
       e <-- e+2 ,
       oldh <-- h-4 ] ,
[      mem(e) <-- oldh ,               % Save old heap pointer on stack
       e <-- e+2 ,
       a1 <-- u ] ,
[      loadlabel(mem(s), back1) ,
       s <-- s+1 ,
       goto([diff]) ] ,                % 'd(U,X,DU)'
[      label(back1) ,
       oldh <-- mem(e-2) ,             % Restore old heap pointer from stack
       e <-- e-2 ] ,
[      a2 <-- mem(e-1) ,               % Restore "x" and "v" from stack
       a1 <-- mem(e-2) ,
       e <-- e-2 ,
       goto([((t\==const@0), true1),
             (dfail)]) ] ,             % Fail if subroutine failed
[      label(true1) ,
       mem(oldh+2) <-- a3 ] ,          % Fill in 1st argument of "++"
[      mem(e) <-- oldh ,               % Save old heap pointer on stack
       e <-- e+2 ] ,
[      loadlabel(mem(s), back2) ,
       s <-- s+1 ,
       goto([diff]) ] ,                        % 'd(V,X,DV)'
[      label(back2) ,
       oldh <-- mem(e-2) ,             % Restore old heap pointer from stack
       e <-- e-2 ,
       goto([((t\==const@0), true2),
             (dfail)]) ] ,             % Fail if subroutine failed
[      label(true2) ,
       mem(oldh+3) <-- a3 ,            % Fill in 2nd argument of "++"
       a3 <-- struct@oldh ,            %  and succeed
       t <-- const@1 ] ,
[      retaddr <-- mem(s-1) ,
       s <-- s-1 ] ,
[      goto([[retaddr]]) ] ,

                       % 'd(U--V,X,DU--DV) :- d(U,X,DU), d(V,X,DV).'
[      label(dmi2) ,
       u <-- mem(a1+2) ,               % Extract "u","v" from input structure
       v <-- mem(a1+3) ] ,
[      mem(h) <-- const@(--) ,         % Create output structure "--"/2
       mem(h+1) <-- const@2 ,
       h <-- h+4 ] ,
[      mem(e) <-- v ,                  % Save "v" and "x" on stack
       mem(e+1) <-- a2 ,
       e <-- e+2 ,
       oldh <-- h-4 ] ,
[      mem(e) <-- oldh ,               % Save old heap pointer on stack
       e <-- e+2 ,
       a1 <-- u ] ,
[      loadlabel(mem(s), back3) ,
       s <-- s+1 ,
       goto([diff]) ] ,                % 'd(U,X,DU)'
[      label(back3) ,
       oldh <-- mem(e-2) ,             % Restore old heap pointer from stack
       e <-- e-2 ] ,
[      a2 <-- mem(e-1) ,               % Restore "x" and "v" from stack
       a1 <-- mem(e-2) ,
       e <-- e-2 ,
       goto([(t\==const@0, true3),
             (dfail)]) ] ,             % Fail if subroutine failed
[      label(true3) ,
       mem(oldh+2) <-- a3 ] ,          % Fill in 1st argument of "--"
```

```
[       mem(e) <-- oldh ,                % Save old heap pointer on stack
        e <-- e+2 ] ,
[       loadlabel(mem(s), back4) ,
        s <-- s+1 ,
        goto([diff]) ] ,                 % 'd(V,X,DV)'
[       label(back4) ,
        oldh <-- mem(e-2) ,              % Restore old heap pointer from stack
        e <-- e-2 ,
        goto([(t\==const@0, true4),
             (dfail)]) ] ,               % Fail if subroutine failed
[       label(true4) ,
        mem(oldh+3) <-- a3 ,             % Fill in 2nd argument of "--"
        a3 <-- struct@oldh ,             %  and succeed
        t <-- const@1 ] ,
[       retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
[       goto([[retaddr]]) ] ,

                           % 'd(U*V,X,DU*V++U*DV) :- d(U,X,DU), d(V,X,DV).'
[       label(dti2) ,
        u <-- mem(a1+2) ,                % Extract "u","v" from input structure
        v <-- mem(a1+3) ] ,
[       mem(h) <-- const@(++) ,          % Create output structure "++"/2
        mem(h+1) <-- const@2 ,
        structh4 <-- struct@h+4 ,
        structh8 <-- struct@h+8 ] ,
[       mem(h+2) <-- structh4 ,          % Both args of "++" are other structs
        mem(h+3) <-- structh8 ,          %  (both "*"/2)
        h <-- h+4 ] ,
[       mem(h) <-- const@(*) ,           % Create first structure "*"/2
        mem(h+1) <-- const@2 ,
        h <-- h+4 ] ,
[       mem(h) <-- const@(*) ,           % Create second structure "*"/2
        mem(h+1) <-- const@2 ,
        h <-- h+4 ] ,
[       mem(h-2) <-- u ] ,               % Fill in 1st argument of second "*"
[       mem(e) <-- a2 ,                  % Save "x" on stack
        e <-- e+2 ,
        oldh <-- h-12 ] ,
[       mem(e) <-- v ,                   % Save "v" and old heap ptr on stack
        mem(e+1) <-- oldh ,
        e <-- e+2 ,
        a1 <-- u ] ,
[       loadlabel(mem(s), back5) ,
        s <-- s+1 ,
        goto([diff]) ] ,                 % 'd(U,X,DU)'
[       label(back5) ,
        oldh <-- mem(e-1) ,              % Restore old heap ptr and "v"
        v <-- mem(e-2) ,
        e <-- e-2 ] ,
[       a1 <-- v ,
        a2 <-- mem(e-2) ,                % Restore "x" from stack
        e <-- e-2 ,
        goto([(t\==const@0, true5),
             (dfail)]) ] ,               % Fail if subroutine failed
[       label(true5) ,
       mem(oldh+6) <-- a3 ,            % Fill in both arguments of first "*"
       mem(oldh+7) <-- v ] ,
[      mem(e) <-- oldh ,               % Save old heap pointer on stack
       e <-- e+2 ] ,
[      loadlabel(mem(s), back6) ,
       s <-- s+1 ,
       goto([diff]) ] ,                % 'd(V,X,DV)'
[      label(back6) ,
       oldh <-- mem(e-2) ,             % Restore old heap pointer from stack
       e <-- e-2 ,
       goto([(t\==const@0, true6),
             (dfail)]) ] ,             % Fail if subroutine failed
[      label(true6) ,
       mem(oldh+11) <-- a3 ,           % Fill in 2nd argument of second "*"
       a3 <-- struct@oldh ,            %  and succeed
       t <-- const@1 ] ,
[      retaddr <-- mem(s-1) ,
       s <-- s-1 ] ,
[      goto([[retaddr]]) ] ,

                  % 'd(U/V,X,(DU*V--U*DV)/(^(V,2))) :- d(U,X,DU), d(V,X,DV).'
[      label(ddi2) ,
       u <-- mem(a1+2) ,               % Extract "u","v" from input structure
       v <-- mem(a1+3) ] ,
[      mem(h) <-- const@(/) ,          % Create output structure "/"/2
       mem(h+1) <-- const@2 ,
       structh4 <-- struct@h+4 ,
       structh16 <-- struct@h+16 ] ,
[      mem(h+2) <-- structh4 ,         % Both args of "/" are other structs
       mem(h+3) <-- structh16 ,        %  ("--"/2 and "^"/2)
       h <-- h+4 ] ,
[      mem(h) <-- const@(--) ,         % Create structure "--"/2
       mem(h+1) <-- const@2 ,
       structh4 <-- struct@h+4 ,
       structh8 <-- struct@h+8 ] ,
[      mem(h+2) <-- structh4 ,         % Both args of "--" are other structs
       mem(h+3) <-- structh8 ,         %  (both "*"/2)
       h <-- h+4 ] ,
[      mem(h) <-- const@(*) ,          % Create first structure "*"/2
       mem(h+1) <-- const@2 ,
       h <-- h+4 ] ,
[      mem(h) <-- const@(*) ,          % Create second structure "*"/2
       mem(h+1) <-- const@2 ,
       h <-- h+4 ] ,
[      mem(h) <-- const@(^) ,          % Create structure "^"/2
       mem(h+1) <-- const@2 ] ,
[      mem(h+2) <-- v ,                % Fill in both arguments of "^"/2
       mem(h+3) <-- const@2 ,
       h <-- h+4 ] ,
[      mem(h-6) <-- u ] ,              % Fill in 1st argument of second "*"
[      mem(e) <-- a2 ,                 % Save "x" on stack
       e <-- e+2 ,
       oldh <-- h-20 ] ,
[      mem(e) <-- v ,                  % Save "v" and old heap ptr on stack
       mem(e+1) <-- oldh ,
       e <-- e+2 ,
      a1 <-- u ] ,
[     loadlabel(mem(s), back7) ,
      s <-- s+1 ,
      goto([diff]) ] ,               % 'd(U,X,DU)'
[     label(back7) ,
      oldh <-- mem(e-1) ,            % Restore old heap ptr and "v"
      v <-- mem(e-2) ,
      e <-- e-2 ] ,
[     a1 <-- v ,
      a2 <-- mem(e-2) ,              % Restore "x" from stack
      e <-- e-2 ,
      goto([(t\==const@0, true7),
           (dfail)]) ] ,             % Fail if subroutine failed
[     label(true7) ,
      mem(oldh+10) <-- a3 ,          % Fill in both arguments of first "*"
      mem(oldh+11) <-- v ] ,
[     mem(e) <-- oldh ,              % Save old heap pointer on stack
      e <-- e+2 ] ,
[     loadlabel(mem(s), back8) ,
      s <-- s+1 ,
      goto([diff]) ] ,               % 'd(V,X,DV)'
[     label(back8) ,
      oldh <-- mem(e-2) ,            % Restore old heap pointer from stack
      e <-- e-2 ,
      goto([(t\==const@0, true8),
           (dfail)]) ] ,             % Fail if subroutine failed
[     label(true8) ,
      mem(oldh+15) <-- a3 ,          % Fill in 2nd argument of second "*"
      a3 <-- struct@oldh ,           %  and succeed
      t <-- const@1 ] ,
[     retaddr <-- mem(s-1) ,
      s <-- s-1 ] ,
[     goto([[retaddr]]) ] ,

                 % 'd(^(U,N),X,DU*N*(^(U,N1))) :- N1 is N-1, d(U,X,DU).'
[     label(dpo2) ,
      u <-- mem(a1+2) ,              % Extract "u","n" from input structure
      n <-- mem(a1+3) ] ,
[     goto([((type(n)?const), ponum),
           (dfail)]) ] ,
[     label(ponum) ,
      goto([((n\=nil), ponuma),
           (dfail)]) ] ,
[     label(ponuma) ,
      n1 <-- n-1 ,                   % 'N1 is N-1'
      mem(h) <-- const@(*) ,         % Create output structure "*"/2
      mem(h+1) <-- const@2 ,
      structh4 <-- struct@h+4 ,
      structh8 <-- struct@h+8 ] ,
[     mem(h+2) <-- structh4 ,        % Both args of "*" are other structs
      mem(h+3) <-- structh8 ,
      h <-- h+4 ] ,
[     mem(h) <-- const@(*) ,         % Create second structure "*"/2
      mem(h+1) <-- const@2 ] ,
[     mem(h+3) <-- n ,               % Fill in 2nd arg of second "*"
      h <-- h+4 ] ,
[       mem(h) <-- const@(^) ,           % Create structure "^"/2
        mem(h+1) <-- const@2 ] ,
[       mem(h+2) <-- u ,                 % Fill in both arguments of "^"
        mem(h+3) <-- n1 ,
        oldh <-- h-8 ,
        h <-- h+4 ] ,
[       mem(e) <-- oldh ,                % Save old heap pointer on stack
        e <-- e+2 ,
        a1 <-- u ] ,
[       loadlabel(mem(s), back9) ,
        s <-- s+1 ,
        goto([diff]) ] ,                 % 'd(U,X,DU)'
[       label(back9) ,
        oldh <-- mem(e-2) ,              % Restore old heap pointer
        e <-- e-2 ,
        goto([(t\==const@0, true9),
             (dfail)]) ] ,               % Fail if subroutine failed
[       label(true9) ,
        mem(oldh+6) <-- a3 ,      % Fill in 1st arg of second "*"
        a3 <-- struct@oldh ,      %  and succeed
        t <-- const@1 ] ,
[       retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
[       goto([[retaddr]]) ] ,

                          % 'd(--U,X,--DU) :- d(U,X,DU).'
[       label(dmi1) ,
        u <-- mem(a1+2) ] ,              % Extract "u" from input structure
[       mem(h) <-- const@(--) ,          % Create output structure "--"/1
        mem(h+1) <-- const@1 ,
        oldh <-- h ,
        h <-- h+4 ] ,
[       mem(e) <-- oldh ,                % Save old heap pointer on stack
        e <-- e+2 ,
        a1 <-- u ] ,
[       loadlabel(mem(s), back10) ,
        s <-- s+1 ,
        goto([diff]) ] ,                 % 'd(U,X,DU)'
[       label(back10) ,
        oldh <-- mem(e-2) ,              % Restore old heap pointer
        e <-- e-2 ,
        goto([(t\==const@0, true10),
             (dfail)]) ] ,               % Fail if subroutine failed
[       label(true10) ,
        mem(oldh+2) <-- a3 ,             % Fill in argument of "--"
        a3 <-- struct@oldh ,             %  and succeed
        t <-- const@1 ] ,
[       retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
[       goto([[retaddr]]) ] ,

                          % 'd(exp(U),X,exp(U)*DU) :- d(U,X,DU).'
[       label(dex1) ,
        u <-- mem(a1+2) ] ,              % Extract "u" from input structure
[       mem(h) <-- const@(*) ,           % Create output structure "*"/2
        mem(h+1) <-- const@2 ,
        structh4 <-- struct@h+4 ] ,
  [     mem(h+2) <-- structh4 ,         % 1st argument of "*" is another struct
        h <-- h+4 ] ,
  [     mem(h) <-- const@(exp) ,        % Create structure "exp"/1
        mem(h+1) <-- const@1 ] ,
  [     mem(h+2) <-- u ,                % Fill in argument of "exp"
        oldh <-- h-4 ,
        h <-- h+4 ] ,
  [     mem(e) <-- oldh ,               % Save old heap pointer on stack
        e <-- e+2 ,
        a1 <-- u ] ,
  [     loadlabel(mem(s), back11) ,
        s <-- s+1 ,
        goto([diff]) ] ,                % 'd(U,X,DU)'
  [     label(back11) ,
        oldh <-- mem(e-2) ,             % Restore old heap pointer
        e <-- e-2 ,
        goto([(t\==const@0, true11),
             (dfail)]) ] ,              % Fail if subroutine failed
  [     label(true11) ,
        mem(oldh+3) <-- a3 ,            % Fill in 2nd argument of "*"
        a3 <-- struct@oldh ,            %  and succeed
        t <-- const@1 ] ,
  [     retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
  [     goto([[retaddr]]) ] ,

                        % 'd(log(U),X,DU/U) :- d(U,X,DU).'
  [     label(dlo1) ,
        u <-- mem(a1+2) ] ,             % Extract "u" from input structure
  [     mem(h) <-- const@(/) ,          % Create output structure "/"/2
        mem(h+1) <-- const@2 ] ,
  [     mem(h+3) <-- u ,                % Fill in 2nd argument of "/"
        oldh <-- h ,
        h <-- h+4 ] ,
  [     mem(e) <-- oldh ,               % Save old heap pointer on stack
        e <-- e+2 ,
        a1 <-- u ] ,
  [     loadlabel(mem(s), back12) ,
        s <-- s+1 ,
        goto([diff]) ] ,                % 'd(U,X,DU)'
  [     label(back12) ,
        oldh <-- mem(e-2) ,             % Restore old heap pointer
        e <-- e-2 ,
        goto([(t\==const@0, true12),
             (dfail)]) ] ,              % Fail if subroutine failed
  [     label(true12) ,
        mem(oldh+2) <-- a3 ,            % Fill in 1st argument of "/"
        a3 <-- struct@oldh ,            %  and succeed
        t <-- const@1 ] ,
  [     retaddr <-- mem(s-1) ,
        s <-- s-1 ] ,
  [     goto([[retaddr]]) ]
].
```

A.7 Automatically-Compiled Benchmark Programs

A.7.1 four queens — Placing Queens on a Chessboard

```
% Modified four queens benchmark.
%
% Benchmark is started with the call:
%       solve( 4, [], _ )

solve(Bs, [square(Bs,Y) | L], [square(Bs,Y) | L]) :- size(Bs).
solve(Board_size, Initial, Final) :-
        newsquare(Initial, Next),
        solve(Board_size, [Next | Initial], Final).

newsquare([], square(1,X)) :- myint(X).
newsquare([square(I,J) | Rest], square(X,Y)) :-
        X is I + 1,
        myint(Y),
        notthreatened(I, J, X, Y),
        safe(X, Y, Rest).

safe(X, Y, []).
safe(X, Y, [square(I,J) | L]) :-
        notthreatened(I, J, X, Y),
        safe(X, Y, L).

notthreatened(I, J, X, Y) :- threatened(I, J, X, Y), !, fail.
notthreatened(I, J, X, Y).

threatened(I, J, X, Y) :-
        (I = X), !.
threatened(I, J, X, Y) :-
        (J = Y), !.
threatened(I, J, X, Y) :-
        (U is I - J),
        (V is X - Y),
        (U = V), !.
threatened(I, J, X, Y) :-
        (U is I + J),
        (V is X + Y),
        (U = V), !.

:-mode((size(N) :- var(N))).
size(4).

:-mode((myint(N) :- var(N))).
myint(1).
myint(2).
myint(3).
myint(4).
```

A.7.2 mu math — Simple Theorem Prover

```
% Modified mu math benchmark.
%
% Benchmark is started with the call:
%       theorem( 2, [m,i,u,i,u], [], _ )

theorem(Depth,[m,i],X,X).
theorem(Depth,[],_,_) :- fail.
theorem(Depth,R,X,[N|Y]) :- Depth>0, D is Depth-1, theorem(D,S,X,Y), rules(S,R,N).

rules(S,R,3) :- rule3(S,R).
rules(S,R,4) :- rule4(S,R).
rules(S,R,1) :- rule1(S,R).
rules(S,R,2) :- rule2(S,R).
rule1(S,R) :- concat(X,[i],S), concat(X,[i,u],R).
rule2([m|T],[m|R]) :- concat(T,T,R).
rule3([],_) :- fail.
rule3(R,T) :- concat([i,i,i],S,R), concat([u],S,T).
rule3([H|T],[H|R]) :- rule3(T,R).
rule4([],_) :- fail.
rule4(R,T) :- concat([u,u],T,R).
rule4([H|T],[H|R]) :- rule4(T,R).

concat([],X,X).
concat([A|B],X,[A|B1]) :- concat(B,X,B1).
```

Bibliography

[CM81] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[DLSM81] Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick W. Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C-30(7):460–477, July 1981.

[Dob87] Tep Dobry. *A High Performance Architecture For Prolog.* PhD thesis, University of California, Berkeley, California 94720, May 1987.

[Fis81] Joseph A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[FLS81] Joseph A. Fisher, David Landskov, and Bruce D. Shriver. *Microcode Compaction: Looking Backward and Looking Forward*, pages 95–102. National Computer Conference, 1981.

[Hay89] Ralph Haygood. *A Prolog Benchmark Suite for Aquarius.* Technical Report UCB/CSD 89/509, University of California, Berkeley, California 94720, April 1989.

[VRss] Peter Van Roy. *The Aquarius Prolog Compiler.* PhD thesis, University of California, Berkeley, California 94720, work in progress.

[War83] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI, Menlo Park, California, October 1983.