The Parallel Programming Language Id and its Compilation for Parallel Machines

Computation Structures Group Memo 313
July 30, 1990

Rishiyur S. Nikhil

To appear in
Proc. Workshop on Massive Parallelism: Hardware, Programming and Applications, Amalfi, Italy, October 1989, Academic Press

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

The Parallel Programming Language Id and its Compilation for Parallel Machines

Rishiyur S. Nikhil

Abstract

In this paper, we show a novel compilation method from a familiar class of languages to a familiar class of architectures. Id is basically a non-strict functional programming language, but it has a computation rule different from the traditional lazy evaluation rule, leading to enormous amounts of parallelism. Then, instead of using graph reduction (the traditional compilation method), we translate Id programs *via* two intermediate languages: first to dataflow graphs, and then to code for an abstract machine called P-RISC. We describe how to implement the P-RISC abstract machine on a message-passing multicomputer, and we describe optimizations at all levels. Throughout, the underlying theme is efficient fine-grained, data-driven execution, which we believe is essential for large-scale MIMD machines. We also remark on the suitability of the approach for other source languages (including FORTRAN).

1 Introduction

Functional languages have always been attractive for their expressive power and clean semantics. However, they have not achieved widespread acceptance to date, in part because their implementations have been slow compared to traditional programming languages. Because traditional languages are so much better matched to sequential computers, there has been little hope that this performance gap would be closed soon (despite excellent work in compiling functional languages [27]). However, for parallel machines, the situation may be reversed, *i.e.*, functional languages may form a better basis for compilation.

Since they do not contain assignments, functional programs do not contain any anti-dependencies [20]; so, they can be partitioned into very fine-grained parallel threads. The parallelism in functional languages is two-fold. First, given a function application:

$f\ arg_1\ \ldots\ arg_n$

it is always safe to evaluate all the arguments in parallel. Second, if the language has *non-strict* semantics, f can be invoked before the arguments are fully evaluated, so that computations in the function body can be overlapped with computation of the arguments. In fact, the function can even return a result before the arguments are evaluated.

Functional programs are also *determinate*. Despite the parallel, non-deterministic scheduling in its implementation (which may depend, for example, on the machine configuration), the answer is always unique. This is an invaluable tool in debugging.

Traditionally, functional languages have been compiled using (1) continuations and optimization of closures [16, 19, 3], or (2) graph reduction[1] [27]. Both approaches were originally developed for sequential machines, but graph reduction compilers have recently been extended for parallel machines [29, 6, 10]. One problem with these approaches is that they exploit parallelism only in *strict* contexts; as we shall see, this can be a serious limitation on parallelism. Another problem is that the abstract machines do not seem to address the issues of long latencies and frequent synchronization that must be present in massively parallel machines.

In contrast, a different approach has been pursued by our group at MIT. From the beginning, it has been motivated by parallelism, and broadly speaking, the framework has always been to compile the functional language Id to dataflow graphs, which are then compiled to machine code for a dataflow machine. However, the language, compilation methods and architectures have evolved significantly over the last ten years.

Originally just a notation for dataflow graphs, Id was redesigned completely in 1985 (when it was briefly called "Id Nouveau"). Today, Id is a full-fledged functional language, with higher-order functions, a polymorphic type system, list and array comprehensions, pattern matching, *etc.* Id has non-strict semantics; however, unlike most other implementations of non-strict languages, we do not use lazy evaluation, and this leads to massive parallelism.

Initially, the only target for Id was the MIT Tagged-Token Dataflow Architecture [5]. Recently, our focus has broadened to include various "multi-threaded architectures", such as Monsoon, proposed by Papadopoulos and Culler [26], and P-RISC, proposed by us [24]. Our work on P-RISC, together with Traub's work on compiling Id [32, 33] for sequential machines, has, for the first time, shown us a clear path towards compiling Id for machines based on traditional von Neumann architectures (both single and multiprocessors).

Figure 1 shows the relationship between all the above compilation strategies. The Tagged Token Dataflow Architecture compiler has been running since 1986 [31]. Together with a highly instrumented software emulator for the TTDA, this experimental platform has been the main vehicle of our research. Today, the compilation effort in our group is mainly directed towards compiling for Monsoon (we are in the advanced stages of actually producing Monsoon prototype hardware, in partnership with Motorola, Inc.) In the last year or so, we have also increased our effort in the P-RISC direction.

A significant difference between all the compilers in Figure 1 and traditional compilers is our approach to synchronization—*all execution is completely data-driven.* Traditional compilers are built on the assumption that asynchronous events are the exception rather than the rule, if they even occur at all. This situation is reversed in massively parallel, scalable architectures [4]; even something as ubiquitous as a memory reference may have to be treated as an asynchronous transaction if the machine is to tolerate the long cross-machine latencies of parallel machines and the synchronization waits due to producer-consumer parallelism.

[1]Recent progress in graph reduction indicates that the two approaches are, in fact, quite similar [28].

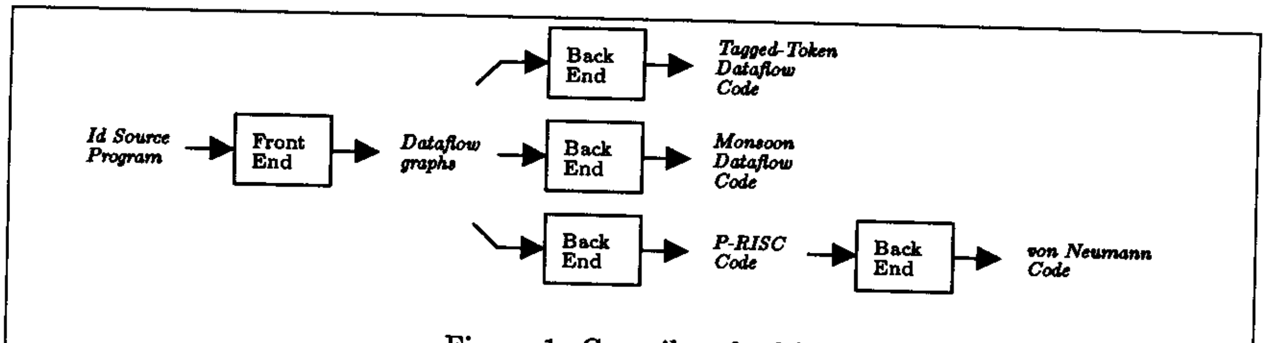

Figure 1: Compilers for Id

By using data-driven execution uniformly, our implementations ensure that synchronizations can be performed frequently, and with *no busy waiting*.

In this paper, we outline a design for the lower path in Figure 1 (the P-RISC back end has not yet been constructed). The paper divided into three major parts. The second and third parts are, in fact, independent of functional languages. They start with dataflow graphs, and it does not matter whether the graphs were produced from Id or from FORTRAN.

The first part deals with the language and sources of parallelism. In Section 2 we briefly describe Id, and in Section 3 we demonstrate the unusual amounts of parallelism in Id programs, comparing it with the parallelism available if we limited it to strict contexts.

The second part deals with compilation to dataflow graphs and the P-RISC abstract machine. In Section 4 we describe *dataflow graphs*, an intermediate language in which fine-grained parallelism is manifest but in which synchronization is implicit; and, we outline how Id programs are translated into dataflow graphs. In Section 5 we describe the P-RISC abstract machine, in which threads and synchronization are explicit. In Section 6, we describe the translation from dataflow graphs into P-RISC, including various optimizations to reduce overhead.

The third part (Section 7) deals with implementation of the P-RISC abstract machine on stock MIMD machines. Finally, we conclude with some comparisons to related work.

2 The programming language Id

Id is a higher-order, typed, functional programming language [25, 23]. In this regard it is similar to ML [22], Miranda [34] and Haskell [17]. Like the latter two languages, Id is non-strict.

Functions

Id programs are organized around user-defined functions. Here is a simple example:

```
def incr y = 1 + y ;
```

which can be read as: "define the increment of y to be y plus 1". Thus, the expression "incr 5" evaluates to 6, and the expression "incr 20" evaluates to 21.

Now, suppose we are given a graph, represented as a list of y-coordinates, and suppose we wish to shift this graph up by one unit, *i.e.,* we wish to increment each y-coordinate by 1. We could use the following function, which takes a list of numbers l and returns a new list whose j'th element is $l_j + 1$.

```
def incr_list Nil     = Nil
  | incr_list (y:l') = (incr y):(incr_list l') ;
```

The two *clauses* of the function take care of the two possibilities for the list argument: either it is empty (Nil), or it is not. If empty, the result is also empty. If not, let y name the head and l' name the tail of the list. Then, we compute the increment of y; we apply incr_list recursively to l' to produce an incremented version of l', and we put them together into a new list using the infix list-construction operator ":". Thus, given a graph g (a list of y-coordinates), we can compute the graph that is the one-unit vertical shift of g by evaluating the expression:

```
incr_list g
```

Higher-order functions

Functions themselves may be treated as values, just like numbers and strings. For example, map_list is a function that, given a list l, produces a new list whose j'th component is $f(l_j)$, where f is itself a parameter of map_list.

```
def map_list f Nil     = Nil
  | map_list f (y:l') = (f y):(map_list f l')
```

The structure is very similar to incr_list, except that instead of applying incr to each element, we apply f. Now, our previous function that incremented all numbers in a list g can be written:

```
def incr_list g = map_list incr g ;
```

Map_list is an example of a *higher-order* function because one of its arguments is itself a function. Such functions have wide applicability, facilitating the reuse of code. For example, suppose we had a function that scaled a number by a factor of two:

```
def scale2 y = 2 * y;
```

Then, we can define a function to scale an entire graph g (a list of y-coordinates) by a factor of two by saying:

```
def scale2_list g = map_list scale2 g ;
```

Thus, map_list has captured the common, recursive structure of incr_list and scale2_list by abstracting out the main difference between them, which happens to be a function (scale2 *vs.* incr). The resulting code is thus more modular.

Currying

Another notational convenience is called *currying*. Suppose we wrote:

```
def scale c y = c * y ;
```

The function `scale` can be applied to a single argument c to produce another function f, where f is the function of one argument that scales by c. For example, the expression:

```
scale 2
```

represents a function of one argument that multiplies its argument by 2. For example:

```
def scale2_list g = map_list (scale 2) g ;
```

Discussion

These features—support for higher-order programming and currying notation—are major factors in giving functional programming its reputation for great expressive power; compared to other languages, programs can be very succinct. Higher-order functions can be regarded as the "power tools" of programming—they amplify effort. As a demonstration of this ability, suppose we are given a list of graphs lg (a list of lists of y-coordinates), and we wish to scale them all by a factor of 3. This is easily expressed by the following expression:

```
map_list (map_list (scale 3)) lg
```

Here, `(scale 3)` is a function that scales a number by 3. Thus, `(map_list (scale 3))` is a function that scales a graph by 3, *i.e.,* it scales all y-coordinates in a graph by 3. And, therefore, the entire expression scales all graphs in lg by 3. Thus, `map_list` allows us to raise our level of thinking, from a function that scales numbers to a function that scales entire graphs to a function that scales collections of graphs, and so on.

3 Parallelism in Id

Parallelism due to non-strictness

Id has non-strict semantics (like Haskell and Miranda, but unlike ML and Scheme). This is a significant source of parallelism. Consider `map_list`, whose definition is repeated here:

```
def map_list f Nil     = Nil
  | map_list f (y:l') = (f y):(map_list f l')
```

Suppose we apply `map_list` to some function f and non-empty list l. As soon as it is verified that l is non-empty, the two subexpressions `(f y)` and `(map_list f l')` may be evaluated in parallel. Let us call the results z and m'.

In a strict implementation, we would return a cons cell containing z and m' *after* the computations for z and m' have been completed. In an implementation supporting non-strict semantics, we do not have to wait for z and m' to be produced—we can produce and return the result cons cell *immediately*. The implementation must ensure:

- That z and m' are ultimately stored in the head and tail of the cons cell that was returned, and
- Any other computation that attempts to read the head or the tail must *block* (*i.e.,* wait) until the corresponding value appears there.

In other words, the *producers* and the *consumers* of the head and tail of the cons cell may run in parallel, provided they can *synchronize* at those slots. The producers, of course, are the computations `(f y)` and `(map_list f l')`, and the consumers are some computations that use the result of this call to `map_list`.

In Id, all computations are initiated in parallel, except that the unfolding is controlled by conditionals. In a conditional expression, neither arm is initiated until the predicate value has been computed, after which the selected arm is initiated. Ken Traub has coined the term "lenient evaluation" to describe this evaluation strategy [32].

Note that a language with strict semantics can have both strict and non-strict implementations. In the latter case, we will achieve as much parallelism as Id does. The only possible "wrong" behavior is rather pleasant—it might actually terminate where the semantics specify non-termination.

Parallelism under composition

The parallelism due to non-strictness and lenient evaluation works very well under composition. Suppose we had the following expression:

```
scale2_list (incr_list l)
```

and, suppose `incr_list` and `scale2_list` individually take time t on lists of length n. Then, in a strict implementation (even if it is parallel), the composition would take time $2t$, because the result of `incr_list` would not be available until time t, after which `scale2_list` would take a further time t.

However, in an implementation supporting non-strictness, the computations of `incr_list` and `scale2_list` can be *overlapped*. The first cons cell from `incr_list` (call it l') can be returned immediately, and `scale2_list` can begin work on it *i.e.,* test l' for emptiness; allocate result cons cell l'' and return it; intitiate computation to scale head of l'; initiate recursive call to scale tail of l', *etc.* As soon as `incr_list` has produced the first head (a number), it can be read and scaled by `scale2_list`. Thus, the list-traversals of `incr_list` and `scale2_list`, instead of being conducted one after the other, can be overlapped, with the `scale2_list` traversal following slightly behind the `incr_list` traversal. In other words, `incr_list` and `scale2_list` can operate in a "pipelined" manner. The total time for the computation, therefore, can be of the order of t instead of $2t$.

Figures 2 and 3 show the "parallelism profiles" of the composite expression, when run on Id World, our Id programming environment. In Id World, the program is compiled to a parallel machine language called *dataflow graphs*, which are described in more detail in the next section. These graphs are executed on a simulator that executes all instructions

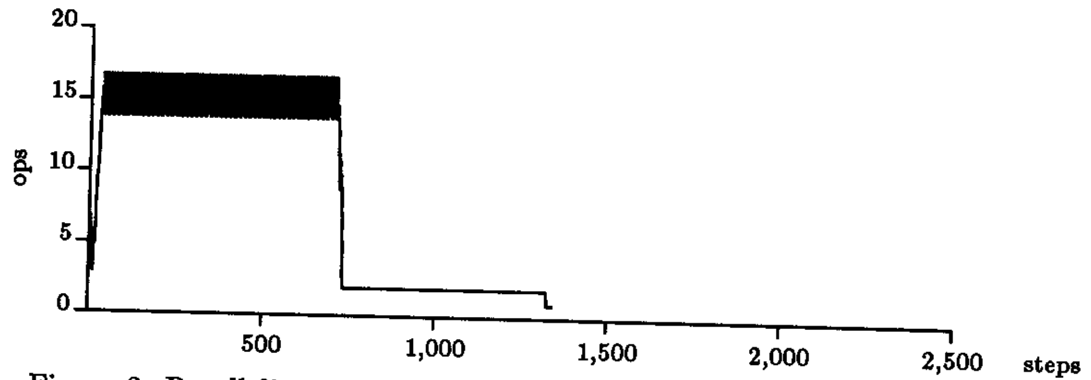

Figure 2: Parallelism profile for non-strict (scale2_list (incr_list l_{100}))

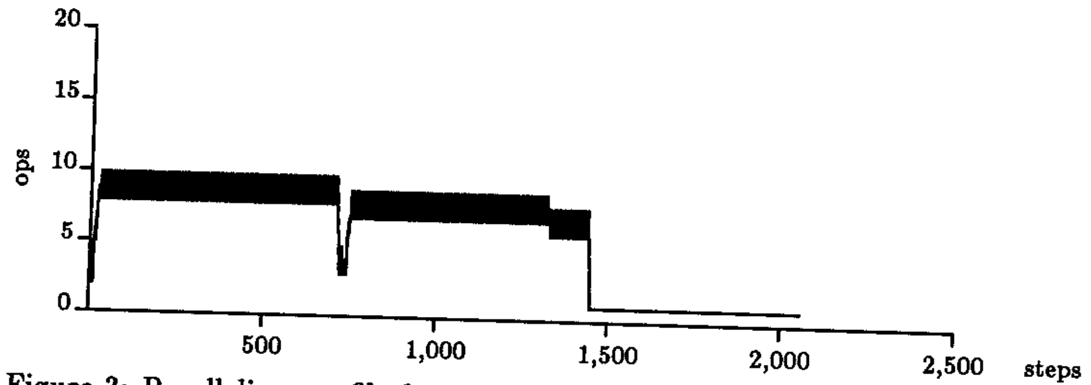

Figure 3: Parallelism profile for strict (scale2_list (incr_list l_{100})) (strict)

in parallel that are ready for execution (*i.e.*, their input data are available). In parallelism profiles, we plot, for each time step (x-axis), the number of operations that are executed in that time step (y-axis).

In both experiments, the expression was evaluated on a list of length 100. Figure 2 shows the normal execution, whereas Figure 3 shows the execution when the interface between `scale2_list` and `incr_list` was artificially constrained to be be strict, *i.e.*, to produce the entire list before any of it can be consumed. In the first case, the program was able to complete in 1343 time steps, with parallelism of the order of 17 (total operations: 11619). In the second case, the program took 2060 time steps, with parallelism of the order of 7 (total operations: 11621). In other words, the non-strict, lenient version was able to complete in roughly half the time, with roughly twice the parallelism.

How much parallelism from non-strict, lenient evaluation?

The parallelism arising out of non-strict, lenient evaluation can be substantial. Consider the following program to compute a list containing the leaves of a binary tree.

```
type tree = Leaf number | Node tree tree ;

def leaves t = aux t Nil ;

def aux (Leaf j)   lvs = j:lvs
 |  aux (Node L R) lvs = aux L (aux R lvs) ;
```

The `type` declaration for trees specifies that a tree is either a leaf containing a number or a node with two sub-trees. The `leaves` function simply calls an auxiliary function, passing it the tree and an empty list of numbers. The `aux` function, given a tree and a list of numbers, concatenates a list of the tree's leaves to the given list. Given a tree that is a leaf with number j, it just attaches j to the given list of leaves. Otherwise, it is given a tree that is an internal node with two subtrees L and R. It concatenates the leaves of R onto the given list, and concatenates the leaves of L onto that list, which gives us the result list.

In a strict implementation, in the second clause of `aux`, the recursive call of `aux` on R must complete before the recursive call of `aux` on L can begin. Thus, the computation becomes a sequential, right-to-left, depth-first traversal of the tree, building the list as it goes. In Id, on the other hand, the recursive call of `aux` on L can begin even though the list produced by the recursive call on R is not available yet. Thus, the traversal of both subtrees can proceed in parallel, producing *exponentially* more parallelism than the strict version.

This is clearly borne out in Figures 4 and 5, which show parallelism profiles for two versions of this program, when run on full binary trees of depth 10. Again, the first profile is the normal one, and the second profile is for a version that was made artificially strict (but still parallel). The non-strict version could be completed in 250 time steps, with a maximum parallelism of 1776 (total instructions: 66,533), while the strict version took 26650 time steps, with a maximum parallelism of 4 (total instructions: 58,349).

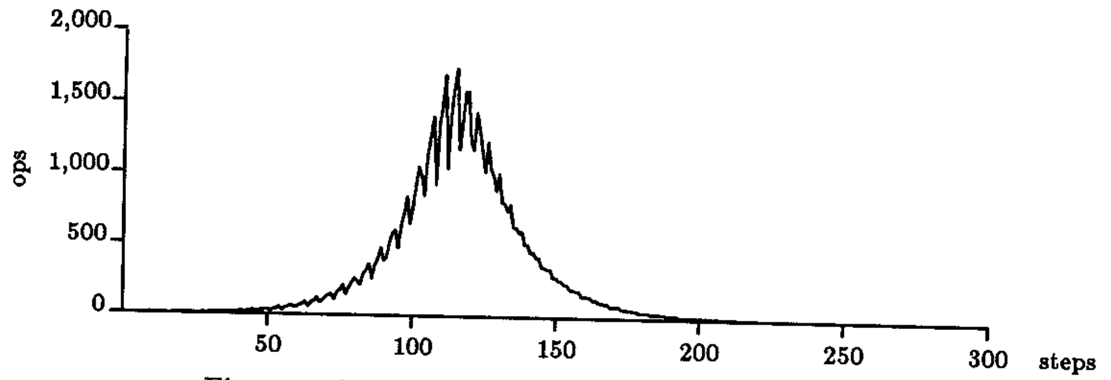

Figure 4: Parallelism profile for non-strict (**leaves** t_{10})

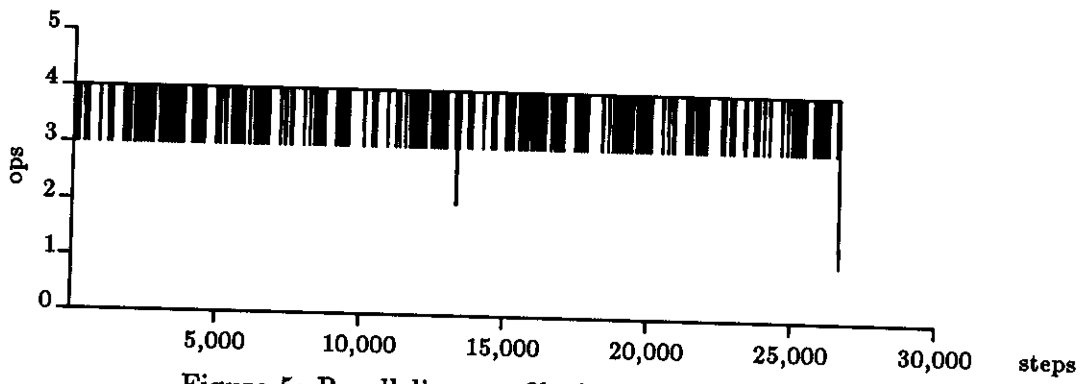

Figure 5: Parallelism profile for strict (**leaves** t_{10})

A note about our parallelism profiles

The profiles shown in this section should be read with care. Since they are computed on a software simulator under idealized conditions, they show only the "maximum" parallelism possible for the particular object code produced by our compiler. Like MIPS ratings, any actual implementation is unlikely to reach this theoretical maximum. Further, the shapes depend somewhat on the compilation strategy. However, the parallelism profiles are useful to gain a first-order estimate about the available parallelism in a program. The profiles shown are for programs that are slightly different from the programs shown in the paper. This is because some extra code had to be introduced to induce the articifial strictness for the strict versions and, therefore, similar dummy code had to be introduced in the non-strict versions in order to balance the total instruction counts. However, the effect of these modifications is not significant.

Relation to lazy evaluation

We can use a common framework for comparing Id's lenient evaluation strategy with the more traditional *lazy evaluation* strategy for non-strict languages. We regard these strategies as different choices for scheduling and synchronization:

- When are processing resources scheduled for producers and consumers?
- How do producers and consumers synchronize?

We use the terms "producer" and "consumer" broadly. A producer may be a computation that writes into a data structure, or one that computes an actual parameter to a function. A consumer may be a computation that reads from a data structure, or one that uses a formal parameter of a function.

In a lazy evaluator, the scheduling of processing resources is as follows. First, devote *all* processor resources to consumers, suspending all producers. When a consumer really needs a particular value, transfer all processor resources to the corresponding producer until it produces a value in weak head normal form, after which all processor resources revert to the consumer. Synchronization manifests itself as a "test for weak head normal form" or, in the terminology of graph reduction, testing to see if the graph already been reduced to a value. Taken literally, this lazy-evaluation scheduling policy results in a completely sequential evaluation strategy (which is perhaps why the "test for WHNF" is not often recognized as a synchronization event). This is clearly a fine choice when there is only a single processor, but what should be done for parallel machines?

Many researchers who are extending graph reduction to parallel machines plan to use strictness information to relax the sequential order of lazy evaluation. Thus, if a function is known to be strict in a particular argument x, then the computation for x is initiated earlier, and in parallel with other computations, rather than waiting for the function to demand it. However, it remains to be seen how effective this approach will be in exposing parallelism, since strictness analysis is known to be very hard in the presence of higher-order functions and data structures. We have not yet seen any experimental results analogous to our parallelism profiles for approaches based on lazy evaluation.

Finally, it is important to realize that the choice between lenient evaluation and lazy evaluation-plus-strictness analysis is manifest *only* in the translation from the non-strict source language into our intermediate language of dataflow graphs. Dataflow graphs and the underlying architectures themselves are neutral to this choice of evaluation strategy.

4 The front end: Id programs to dataflow graphs

In traditional compilers, the source program is first translated into a *control flow graph*, which is a graph whose nodes are either 3-address instructions or basic blocks and whose arcs represent the flow of control due to unconditional and conditional jumps [2]. Through data flow analysis, the compiler may superimpose new edges that depict the *data dependences* from one node to another. The control flow graph may be further reorganized into a *control dependence graph* which, roughly speaking, ties each node to a conditional node that controls its execution. Recently, Ferrante *et. al.* proposed the *program dependence graph*, or PDG [13] as a unified intermediate form for parallel compilers. A PDG contains the control dependence graph with data dependence edges superimposed on it, and compilers for parallel machines can use PDGs to reorganize the statements into parallel threads. The statements are partitioned into threads that respect the data dependencies, either by explicit sequencing or by the introduction of synchronization primitives such as semaphores.

Our approach to compiling Id is quite different. The source program is first translated into a *dataflow graph*, which depicts only data dependences. In other words, the dataflow edges *are* the control flow edges—they describe the minimum constraints on the order in which operators must be executed. A node in a dataflow graph is not a conventional three-address instruction—it specifies only an opcode, and does not specify sources or a destination. Its sources are considered, implicitly, to be the values produced by its predecessors in the dataflow graph, and its output value is considered, implicitly, to be a source for operators that are its successors in the dataflow graph. The actual allocation of locations for these sources and destinations is postponed to later phases of the compiler.

Simple expressions

For simple expressions, the dataflow graph looks very similar to the parse tree (by tradition, dataflow graphs are drawn so that data usually flows downward). Figure 6 shows the parse tree and dataflow graph for the expression `(a+b)*(a-b)`. Note that multiple uses of a variable are made explicit.

A dataflow graph specifies the minimum order in which operations must be performed. Once `a` and `b` are available, either the addition or the subtraction, or both, may be performed. Once the sum and difference are available, the multiplication can be performed. Note that it is only a *partial order* on operations, giving us the flexibility of executing various operations in parallel. Also, note that the arcs entering a dataflow graph are labelled by the free variables of the expression, and the arc leaving the graph represents the value of the expression.

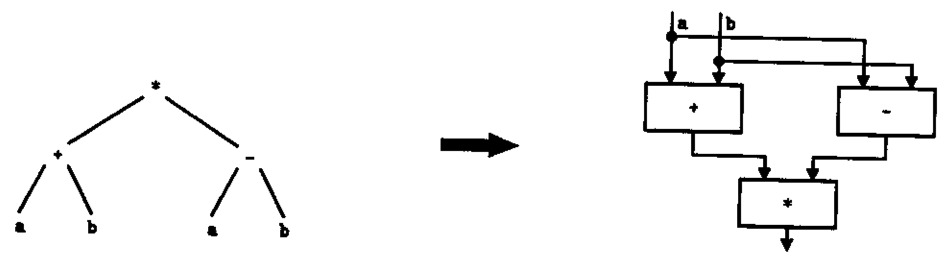

Figure 6: Conventional and dataflow graph representations of a simple expression

Synchronization and state in dataflow graphs

There are two kinds of synchronization in dataflow graphs, and both are *implicit*. The first is called a *join* synchronization, and is present for all operators with more than one input arc, such as the three operators of Figure 6. Each such operator involves synchronization because the operation should not be scheduled until all its predecessors have been executed, *i.e.*, we need some way to detect that all its predecessors have completed.

The second kind of implicit synchronization is tied with state; in particular, with accesses to data structures. In our model, a dataflow graph is always executed with respect to a *heap memory*. An arc representing a data structure in fact represents a pointer into the heap, where all data structures reside. We have a repertoire of operators to store and retrieve components of heap data structures. For example, the "`store` j" operation takes a pointer A and a value v and stores v in heap location $A+j$ (here, j is assumed to be a small constant offset, which is why we include it in the operator). Similarly, the "`fetch` j" operation takes a pointer A and returns the contents of heap location $A+j$.

Two of the heap operations are synchronized. The "`I-store` j" and "`I-fetch` j" operations are similar to `store` j and `fetch` j, respectively. However, the heap location is assumed to be initially marked empty. When an `I-store` is performed on it, it is marked full. If an `I-fetch` is attempted against an empty location, it automatically blocks until it becomes full, after which the value is returned. No more than one `I-store` may be performed on a location (it is a runtime error, otherwise), but any number of `I-fetch`'es may be performed. This synchronization on heap accesses is very useful in compiling functional languages, where a heap location may be written to at most once. It gives us the flexibility to safely schedule the producer (with an `I-store`) and consumers (with `I-fetch`'es) in parallel. In fact, this is also useful in compiling other languages in the situations where we can detect this producer-consumer behavior.

Triggers and signals

Because of non-strictness, it is often possible for some computation in an expression to be initiated even before any of its "normal" inputs are available. Consider the list-construction expression `(a:b)`. Here, we can allocate the cons cell even before `a` or `b` is available. To accomodate this, we often augment a graph with an arc that is called the *trigger* input.

This input can be interpreted as a dummy value that is produced early, whose only role is to enable computations that do not require any other input.

Similarly, because of non-strictness, it is often possible that a result is produced from a dataflow graph before all computations in the graph have terminated. Again, in the expression `(a:b)`, we can return a reference to the newly allocated cons cell before we have stored `a` and `b` into the head and tail slots. This becomes an issue if we want to reclaim certain resources allocated for the graph. For example, suppose we wish to deallocate an activation record (frame) for a procedure. When is it safe to do so? Only when *all* its computations have terminated, which may occur later than the time that it returns a result. For this reason, we augment the basic dataflow graph for an expression with a new output called a *signal*. This output can be interpreted as a dummy value such that when both the signal and the normal output have been produced, we are guaranteed that all computation in the graph has terminated.

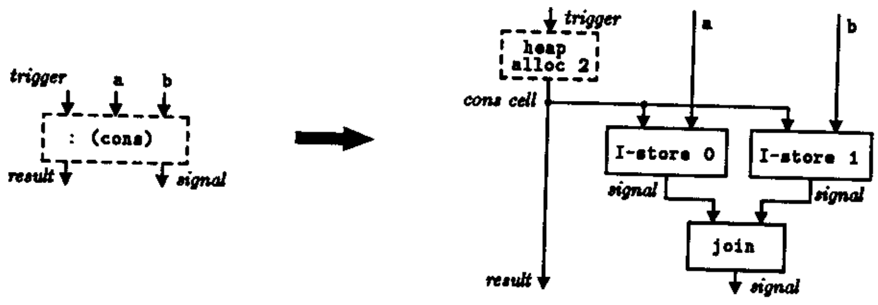

Figure 7: Dataflow graph for (a:b), including triggers and signals

Figure 7 shows the complete graph for the expression `a:b`. By convention, we will use dashed lines for non-primitive operators, *i.e.,* abbreviations for more detailed dataflow graphs. As soon as the `(heap alloc 2)` is triggered, it allocates a cons cell (chunk of heap memory of size 2), and produces a pointer to this cell on its output arc. This pointer is immediately returned as the result of the graph, as well as sent to the left inputs of the two `I-store` operators. When the inputs `a` and `b` arrive, the `I-store` operator can execute, storing them in heap memory at offsets 0 and 1, respectively, relative to the pointer to the cons cell. Each I-store operator produces a signal, indicating that it has executed. The `join` operator produces a signal when it receives all its inputs, *i.e.,* the two I-store signals. It can easily be verified that when the final result and signal are produced, all operators in the graph have executed.

Conditionals

Consider again our `map_list` procedure:

```
def map_list f Nil    = Nil
  | map_list f (y:l') = (f y):(map_list f l')
```

The dataflow graph for the body of the procedure is shown in Figure 8. The graph has three inputs: a trigger, `f` and `l`. The list `l` is tested to see if it is equal to `Nil`, producing a boolean value that goes to each of the three switch operators. The switches direct the three inputs (trigger, `f` and `l`) either to the left or to the right, depending on whether the boolean value is true or false, respectively.

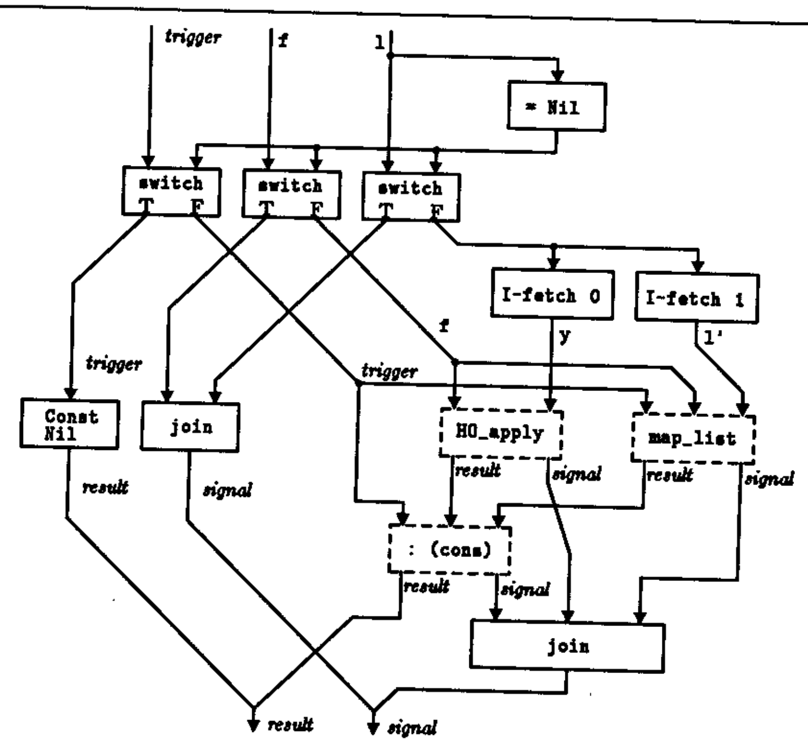

Figure 8: Dataflow graph for body of `map_list`

If true (`l` = `Nil`), the trigger produces a constant `Nil`, and the other two inputs (`f` and `l`) are gathered through a `join`. Note that the `join` is necessary because otherwise there is no guarantee that the second and third switches have executed, even after `Nil` has been produced as the result.

If false (`l` ≠ `nil`), then the two `I-fetch` operators are used to fetch the head `y` and the tail `l'` of the list from heap locations at offsets 0 and 1, respectively, from the pointer `l`. The function `f` is applied to `y` using the higher-order application operator `HO_apply`, and `f` and `l'` are passed to a recursive invocation of `map_list`. The two results are cons'ed together to produce the final result of the else side. The results and signals from the then and else sides are merged together at the bottom of the graph.

Note the use of triggers. When `l` is empty, the trigger allows us to return the constant `Nil` even if `f` is not yet available. When `l` is non-empty, the trigger allows us, even before `f` and `y` and `l'` are available, to:

(a) allocate and return the result cons-cell; and
(c) initiate the recursive call to `map_list`.

We have already seen how (a) is achieved, and we shall shortly see how (c) is achieved. Note also the production of signals, especially those coming out of the bottom right-hand sides of the cons, HO_apply and map_list subgraphs. The reader should convince himself that, assuming that these subgraphs produce proper signals, then the entire graph produces a proper signal.

Procedures and procedure calls

Figure 9 shows the packaging of the dataflow graph of an expression that converts it into a procedure. The three id ("identity") operators at the top receive return information and the two arguments, respectively. Receipt of the return information is used as a trigger which, along with the arguments, is fed into the procedure body. The dashed box for the procedure body represents the graph of Figure 8. Finally, the result and signal of the procedure are returned to the caller using the return information. Note that the packaging accomodates non-strict procedures. In particular, it is possible for the procedure to return a result at any time after the return information has arrived, even if none of the arguments have arrived.

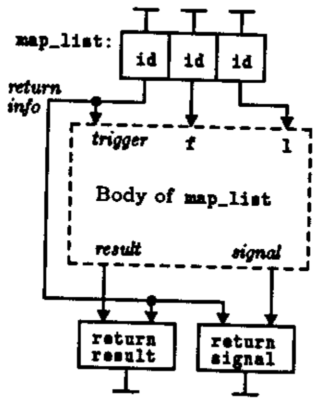

Figure 9: Packaging the dataflow graph of an expression to make it a procedure.

To support procedure calls, we need to name certain nodes in a graph. We do this by writing a label "*L:*" just outside the node. For example, in Figure 9, we have labelled the first node with "map_list:". Of course, node labels should be unique. In the figures, we will label nodes that are entry points of procedures, and nodes that are return points, where results of procedure calls are received.

Since we support higher-order procedures, the identity of the procedure being called may not be known statically at a call site. As we shall soon see, at run time we will know the label of the first entry point of the procedure. Thus, we also need a way to determine the labels of all the remaining entry points of a procedure from this one. If Figure 9, we juxtapose the three id operators at the top of the graph to suggest that the labels of all three entry points are derivable from the label of the first one. Later, when we look at P-RISC code, we shall see that these will correspond to adjacent instructions in a linearly-addressed memory.

First-order procedure applications

For procedure calls, we distinguish between first-order and higher-order applications. An application is first-order if (a) if we statically know the procedure being called, and (b) it is not curried, *i.e.,* it is applied to as many actual parameters as there are formal parameters. Otherwise, an application is higher-order. An example of a first-order application is the recursive call to `map_list` inside the body of `map_list`. On the other hand, the application `(f y)` inside `map_list` is a higher-order application. A first-order application is translated as shown in Figure 10. The "`const map_list`" operator produces the label for the entry-point of the `map_list` procedure.

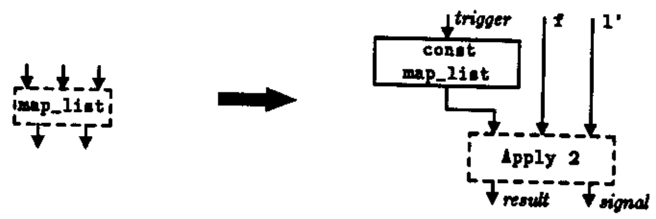

Figure 10: First-order function linkage.

The expansion of the "`apply` n" graph is shown in Figure 11, for $n = 2$. It takes an entry label of a procedure and 2 arguments; invokes that procedure on those arguments, and returns the result and the signal from the procedure. From the label, we determine the size of the frame required for the procedure. This can be done using some convention; for example, the compiler can store the frame size for a procedure at some known offset from the entry label for the procedure. Then, we allocate a frame of the required size, producing a reference to this new frame, `FP'`. Using this, we send the return information and the two arguments to the procedure. The return information will include, among other things, a reference to the current frame and the label L to return to. Again, we juxtapose the two identities that receive the result and the signal to suggest that the callee knows both of them from the label L. In P-RISC code, they will be adjacent instructions in a linearly addressed code memory.

We have been deliberately vague by mentioning "frames", "frame sizes" and "frame pointers" without specifying exactly what frames *are*. In this section, we are only trying to appeal to the reader's intuition from sequential machines to convey the high-level idea that we, too, will distinguish multiple invocations of the same piece of code by giving each invocation a different frame, or activation record. We will complete the picture by describing frames in detail when we describe P-RISC code and its implementation.

Higher-order procedure applications

Higher-order procedure applications are compiled using a simple transformation that converts them into first-order applications. Consider the `scale` function from Section 2:

```
def scale c y = c * y ;
```

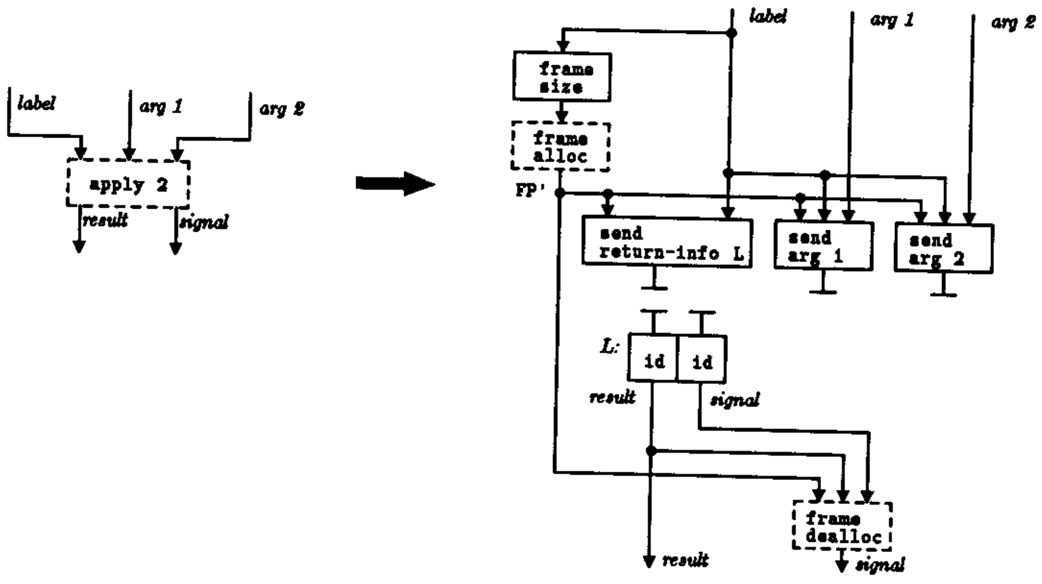

Figure 11: Application of a procedure with two arguments.

Consider any occurrence of the identifier `scale` in the program where it appears in a higher-order context, *i.e.*, where it is not being applied to two arguments. We replace that occurrence of the identifier `scale` by:

```
(const scale0 , Nil)
```

whose dataflow graph is shown in Figure 12. In essence, we are creating a *closure*, which is a pair—a procedure label and an environment. In this case, the label is the entry point of procedure `scale0` (to be described momentarily), and the environment is empty (`Nil`). Note: the notation "`const` *scale0*", which designates a procedure label, is not a part of the source language.

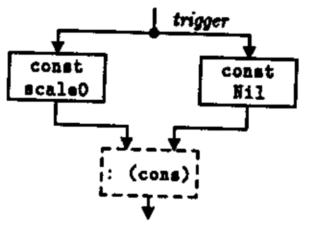

Figure 12: Initial creation of a closure

Each higher-order application involves the application of a closure `f` = (L, env) to an argument `y`. We convert it into a first-order application that invokes L on env and `y`, as shown in Figure 13. Now, we compile two more functions that are derived from the original `scale` function (the dataflow graphs are quite trivial, so they are not shown):

```
def scale0 env c = (const scale1, (c:env)) ;
```

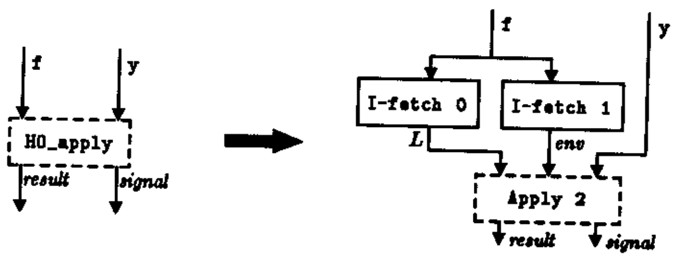

Figure 13: Higher-order application

```
def scale1 env y = { (c:Nil) = env
                     In
                       scale c y } ;
```

Thus, initially, we create a closure containing the label `scale0` and an empty environment. When this is applied to an argument `c`, it simply creates a new closure containing the label `scale1` and an environment which extends the previous environment with this argument. When this closure, in turn, is applied to an argument `y`, it simply extracts `c` from the environment and performs an ordinary first-order apply of `scale`. Note that, because of the non-strictness of tuples and the cons operator, higher-order procedure applications retain all the non-strictness of first-order applications.

Other translations, optimizations

The Id compiler also includes translation schemes for loops and case expressions. It performs a variety of conventional optimizations, such as common subexpression elimination, removal of redundant `I-fetch`'es, signals and triggers, removal of dead code, inline substitution, special representations for closures and applications, optimizations of array index computations, code hoisting *etc.* These are described in more detail in [31].

By recognizing how higher-order functions are actually used in a particular program, there is much room for improvement over the general methods described above. For example, suppose we have a function of 5 arguments:

```
def f a b c d e = ... ;
```

and let us further suppose that there is only one higher-order occurrence of `f` in the program:

`f` x_1 x_2 x_3 x_4

Then, that occurrence can be replaced by:

`(const` *f4′* `: (`x_1`,`x_2`,`x_3`,`x_4`))`

and we can define the derived function:

```
def f4' env e = { (a,b,c,d) = env
                  In
                    f a b c d e } ;
```

We have only one derived function (instead of a chain of derived functions), and we represent the environment as a flat 4-tuple instead of a list of four values. Thus, the higher-order calling convention can be tailored to suit each use in a program.

One major class of optimizations in the Id compiler concerns loops. In conventional tail recursion optimization, a single frame is reused from one iteration to the next. However, this is too sequential, since one iteration must complete before the next one begins. In the Id compiler, we generalize this to allow k iterations to run in parallel, for some integer k (unrelated to the loop bounds). We pre-allocate k frames and initiate the first k iterations. When the 1'st iteration completes, it passes its frame to the $k+1$'st iteration; when the 2'nd iteration completes, it passes its frame to the $k+2$'nd iteration, and so on. Thus, during the entire loop, there is no interaction with the frame allocator at all. The parameter k can be determined dynamically. Further, through analysis, the aim is to completely eliminate all "consing" from a loop, with no loss of parallelism. These optimizations make use of lifetime analysis of data structures, loop unrolling and loop peeling, loop bounding, *etc.*, and will be described in detail by James Hicks in his forthcoming Ph.D. thesis [15].

Dataflow graphs constitute a complete, executable, fine-grained parallel machine code. Optimizations at this level are cleaner because dataflow graphs avoid a certain degree of clutter present in other intermediate representations, such as variable names in 3-address code. Further, they make *def-use* information apparent and explicit.

5 The P-RISC abstract machine

Dataflow graphs still hide certain machine-level details:

- Where should the values produced by each operator reside, (*i.e.*, how do values "flow" down arcs?
- How should the "join" synchronization of dyadic operators be implemented?
- How should the `I-fetch` operators block on empty heap locations?

In this section we describe an abstract machine called P-RISC (for "parallel RISC") where these details are explicit. The picture will be completed in the next section, when we describe the translation from dataflow graphs to P-RISC code. P-RISC was first proposed in [24] as a concrete architecture; here we simplify it into an abstract machine, as shown in Figure 14.

Each token in Token Memory consists of an instruction pointer IP and a frame pointer FP. IP identifies an instruction in Code Memory, which is a conventional array of instructions. FP identifies a frame (or activation record) in Frame Memory. Each invocation of a procedure is associated with a new frame. Finally, all data structures are allocated in Heap Memory. Heap locations have extra state bits so that each location can be marked empty or full. We will use the notation `Frames[j]` and `Heap[k]` to refer to frame and heap locations, respectively.

The machine is operated by repeatedly removing a token from Token Memory and executing the instruction that it points to. This results in a state transition on Frame and the

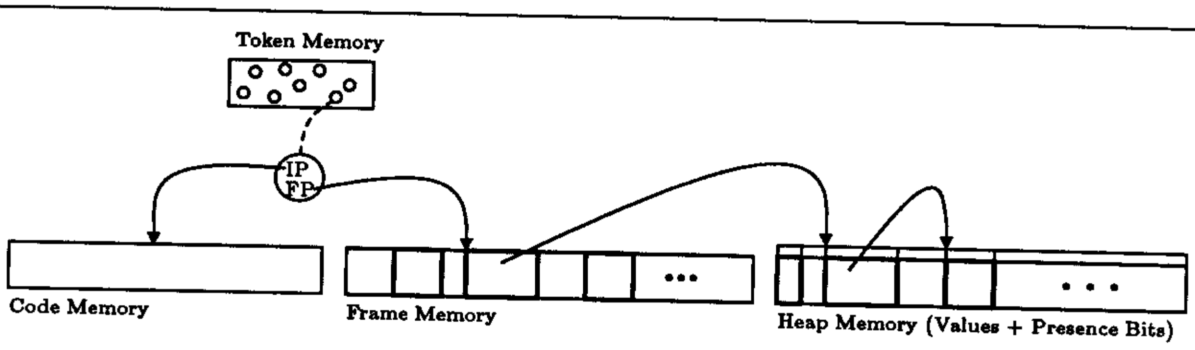

Figure 14: The P-RISC abstract machine model.

Heap Memory, followed by the insertion of zero or more new tokens into Token Memory. The abstract machine does not specify any order in which tokens from token memory are executed. In particular, there is no restriction on how many tokens are executed in parallel.

To relate it to conventional runtime organizations: Code, Frame and Heap memories correspond to traditional code, stack and data segments. Unlike a traditional stack, we have a *tree* of frames because of parallel invocations. Each token can be regarded as a process/task/thread descriptor. Its IP specifies the next instruction to be executed in that thread, and its FP specifies the context for that thread. Most conventional thread models associate a thread with its own stack, so that there is only one thread per frame, indeed only one thread per stack of frames. We place no such restriction, *i.e.*, there may be an arbitrary number of tokens referring to the same frame. This gives us a much finer-grain notion of a thread. Some threads will be barely a few instructions long.

P-RISC Instruction set

In the discussion below, we describe the state transitions when executing a token. Let IP and FP be its instruction and frame pointers, respectively. We will use "`r`" to refer to a frame-relative address, *i.e.*, to frame location `Frames[FP+r]`. Each instruction may involve multiple reads and writes from various memory locations. Unless explicitly specified, there are no atomicity constraints on these sets of accesses.

Arithmetic-Logic

Arithmetic and logical instructions are conventional 3-address instructions.

Notation:	`r1 <- r2` *binop* `r3`
Semantics:	`Frames[FP+r1] := Frames[FP+r2]` *binop* `Frames[FP+r3]`
	Insert token `(FP,IP+1)` in token memory

Notation:	`r1 <-` *unop* `r2`
Semantics:	`Frames[FP+r1] :=` *unop* `Frames[FP+r2]`
	Insert token `(FP,IP+1)` in token memory

Notation:	`r1 <-` *c*
Semantics:	`Frames[FP+r1] :=` *c*
	Insert token `(FP,IP+1)` in token memory

for various *op*'s such as "+", "-", ..., "`negate`", ..., "`and`", ..., "`not`", ... *etc.* There is no atomicity constraint between the reads and writes.

Control

An unconditional jump:

Notation:	`jump L`
Semantics:	Insert token `(FP,L)` in token memory

A simple variant is an indirect jump, `ijump r`, which picks up the destination label from frame location `FP+r`.

A conditional jump:

Notation:	`j`*cond* `r L`
Semantics:	If the condition holds for `Frames[FP+r]`
	Insert token `(FP,L)` in token memory
	Else
	Insert token `(FP,IP+1)` in token memory

for various conditions *cond* such as zero, not-zero, even, odd, ...

A fork:

Notation:	`fork L`
Semantics:	Insert two tokens `(FP,L)` and `(FP,IP+1)` in token memory

Note that a fork does not involve any new frame allocation, *i.e.,* it is of no more complexity than a jump instruction, and there can be multiple threads within a single invocation of a procedure.

A join:

Notation:	`join r`
Semantics:	`Frames[FP+r] := Frames[FP+r] - 1`
	If `Frames[FP+r]` is zero
	Insert token `(FP,IP+1)` in token memory
	Else
	Insert *no* token in token memory

The decrement of the frame location must be performed *atomically*, *i.e.,* two `join`'s on the same location must not see the same value.

Procedure linkage

When one procedure calls another, the following actions must occur:

1. A new frame must be allocated for the callee.
2. Argument values must be moved from the caller's frame to the callee's frame.
3. The caller must initiate one or more threads of computation in the callee.

Similarly, for a procedure return,

4. The callee must move the result value to the caller's frame.
5. The callee must initiate one or more threads in the caller's frame.
6. The callee's frame must be deallocated.

In general, the caller's and callee's frames may be on different processors in a parallel machine. Thus, we use special instructions for this linkage. The 2-3 sequence and the 4-5 sequence are each carried out by `fstore` instructions followed by `fjump` instructions.[2] Here is the description of these instructions.

Inter-frame transfer of data:

Notation:	`fstore r1 r2`
Semantics:	Let `FP'` = `Frames[FP+r1]`
	Let `v` = `Frames[FP+r2]`
	`Frames[FP'] := v`
	Insert token `(FP,IP+1)` in token memory

Inter-frame transfer of control:

Notation:	`fjump r1 r2`
Semantics:	Let `IP'` = `Frames[FP+r1]`
	Let `FP'` = `Frames[FP+r2]`
	Insert token `(FP',IP')` in token memory

One of the arguments sent to the callee is usually the caller's frame pointer (so that the callee knows where to send the result back to). Thus, the caller needs access to its own frame pointer; the following instruction achieves this:

Notation:	`r <- current_FP`
Semantics:	`Frames[FP+r] := FP`
	Insert token `(FP,IP+1)` in token memory

[2]The instructions `fstore` and `fjump` were combined into a single `start` instruction in [24].

Heap access

Ordinary loads and stores move data between the heap and frames (frame location `FP+r2` contains a heap address):

```
Notation:   r1 <- load r2
Semantics:  Frames[FP+r1] <- Heap[Frames[FP+r2]]
            Insert token (FP,IP+1) in token memory

Notation:   store r1 r2
Semantics:  Heap[Frames[FP+r2]] <- Frames[FP+r1]
            Insert token (FP,IP+1) in token memory
```

Synchronized loads and stores have behaviors that depend on the full/empty state of the heap location. When a heap location is empty, it contains a list of triples, where each triple consists of a frame pointer, an instruction pointer and a frame offset. All freshly allocated heap locations are marked empty, and contain an empty list of triples.

```
Notation:   r1 <- i-load r2
Semantics:  Let A = Frames[FP+r2]
            Case Heap[A] of
             (Empty,l)  =>  Heap[A] := (Empty, (FP,IP+1,r1):l)
             (Full,v)   =>  Frames[FP+r1] := v ;
                            Insert token (FP,IP+1) in token memory

Notation:   i-store r1 r2
Semantics:  Let v = Frames[FP+r1]
            Let A = Frames[FP+r2]
            Case Heap[A] of
             (Empty,l)  =>  Heap[A] := (Full,v)
                            For each (FP',IP',r) in l
                              Frames[FP'+r] := v
                              Insert token (FP',IP') in token memory
             (Full,w)   =>  error
```

Note that in each case, the heap location is both read and written. This must be performed atomically.

Memory management

Frame allocation and deallocation:

Notation:	`r1 <- falloc r2`
Semantics:	Allocate a new frame of size `Frames[FP+r2]` Let `FP'` be the address of this new frame `Frames[FP+r1] := FP'` Insert token `(FP,IP+1)` in token memory

Notation:	`fdealloc r`
Semantics:	Let `FP' = Frames[FP+r]` Deallocate the frame at `FP'` Insert token `(FP,IP+1)` into token memory

Heap allocation:

Notation:	`r1 <- halloc r2`
Semantics:	Allocate a new heap object of size `Frames[FP+r2]` Let `A` be the address of this new object `Frames[FP+r1] := A` Insert token `(FP,IP+1)` in token memory

It is assumed that when a heap object is allocated, all its locations are in the empty state. There is no instruction for deallocating heap objects in the abstract machine. Any implementation of the abstract machine needs to have an underlying storage reclamation mechanism, such as a garbage collector.

The list of instructions above is not complete. Clearly, we will need additional instructions to describe frame and heap allocation in more detail, for resetting newly allocated heap locations to the empty state, for garbage collection, *etc.* However, since our focus in this paper is on the main compiler and not on resource management, we stop at the `falloc`, `fdealloc` and `halloc` abstractions.

P-RISC as a model of MIMD machines

We expect that in order to run general-purpose programs on large-scale MIMD machines, it is essential to compile them into a large number of threads (many more threads than the number of physical processors available). The reason is that at any point in time, many threads will be suspended, either because they are waiting for a synchronization event or because they are waiting for some data from some remote part of the machine (as argued by Arvind and Iannucci in [4], these two reasons for suspension are closely related). In order to avoid idling, therefore, each processor should have an adequate pool of threads so that it is likely that at least one is always ready to execute. Further, more threads give more flexibility in balancing the load across processors. The P-RISC abstract machine supports this requirement for numerous, fine-grained threads. One can view a token `(FP,IP)` in token memory as a descriptor for a thread beginning with the instruction at `IP`. The thread includes the successors of that instruction, and terminates when it reaches an instruction such as a `join` or an `i-load`, each of which may require switching to another thread.

A second useful requirement for a parallel machine is that synchronization should be performed without busy-waiting. In the P-RISC abstract machine, synchronization occurs at a `join` or an `i-load`. In each case, there is no busy-waiting—the `join` thread simply dies, and the `i-load` continuation is immediately queued in heap memory, to be awakened exactly once when the location becomes full.

Of course, there are other important requirements for parallel machines, such as fast thread switching, load balancing, *etc.* We will discuss these after describing the mapping of the P-RISC abstract machine to a concrete architecture.

6 The back end: dataflow graphs to P-RISC code

The translation of dataflow graphs to P-RISC code is performed on a procedure-by-procedure basis. For each procedure, we annotate its dataflow graph with P-RISC frame slots and P-RISC code labels; then, generating P-RISC code is straightforward. Simultaneously, we compute the size of the frame required for the procedure, and this is stored along with the P-RISC code at some fixed offset from its entry point. Whereas, in dataflow graphs one thinks of data "flowing" along the arcs, after the annotation we regard the arcs as pure control information, *i.e.*, specifying the order of operations.

We begin by showing a simple, if naive, translation, in order to establish the basic ideas. Here, we pay no attention to the size of the code generated, nor to the locality behavior of the code, nor to limiting the size of the frame. Towards the end of the section, we will discuss more sophisticated translations and optimizations.

Annotation of the DFG

Figure 15 shows an example of a dataflow graph annotated with operator labels, value slots, value arc labels, and join slots.

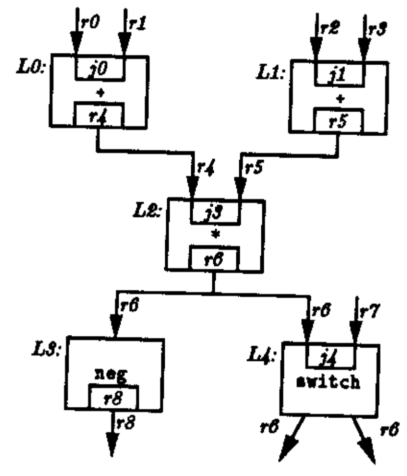

Figure 15: Dataflow graph fragment with PRISC annotations

Operator labelling

We introduce a new label $L0$, $L1$, ... , for each operator in the dataflow graph. Later, when we generate P-RISC code, each operator expands into one or more P-RISC instructions. The label for a dataflow graph node represents the instruction address of the first P-RISC instruction of the node's translation.

Value slot allocation and arc labelling

For every value-producing operator in the DFG, we reserve a frame slot $r1$, $r2$, ..., for the output value of the operator. Thus, we reserve frame slots for the outputs of arithmetic and logic operators, `I-fetch`'es, `const`'s, `heap alloc`'s, `frame alloc`'s, the `id` operators at procedure entries and returns, *etc.* We do not reserve any frame slots for control operators such as `switch`'es, `join`'s, `I-store`'s *etc.*, because these do not produce any new output values.

Every "value-carrying" arc (an arc that is not a signal or trigger) can now be labelled with the frame slot where that value resides, as follows. All output arcs of a value-producing operator are labelled with the frame slot for that operator's output. For `switch`'es, both outputs are labelled with the label of its left input arc, which represents the value being switched. The result arcs of the `then` and `else` sides of a conditional must have the same frame slot label, *i.e.,* both sides should leave the result in the same frame slot (it is easy to ensure this, if necessary by introducing an extra identity instruction on one side).

Join slot allocation

For every operator with more than one input arc, we reserve a frame slot $j1$, $j2$, ..., where we will conduct the "join"-synchronization for that operator. Note that the `neg` operator has no join slot, since it is unary and and therefore does not have to synchronize on the completion of multiple threads. A small optimization that we can perform immediately is this: for value-producing operators, it is perfectly all right for the join slot to be the same as the output value slot. For example, in Figure 15 we can have $j0 = r4$. This is because the join slot is always used before the output value slot is needed.

P-RISC code generation from labelled dfgs

Once the dataflow graph is annotated, translating it to P-RISC code is straightforward. Consider the "`*`" operator in the middle of Figure 15. The P-RISC code for that node is:

```
L2: join j3
    r6 <- r4 * r5
    fork L4
    jump L3
```

Similar code can be generated for each arithmetic/logic operation. We assume that the frame location $j3$ has been initialized to 2, since two threads join there. Thus, we are assured that the neg at node $L3$ and the switch at node $L4$ will not execute until the product has been stored in $r6$. By induction, we are also assured that the multiplication cannot occur until both the preceding additions (in nodes $L0$ and $L1$) have completed.

In general, the join slot has to be initialized to n when the dataflow operator has n inputs ($n \geq 2$). Similarly, if the operator has m successors, the P-RISC code has $m - 1$ fork's followed by a jump. In subsequent examples, we will not explicitly mention the join location j, and we will abbreviate the trailing forks and jump with ellipses.

A switch operator whose true and false outputs go to nodes labelled M and N, respectively, is translated into:

```
L: join ...          % wait for both inputs to be ready
   jtrue r M         % assume boolean value in r
   jump N
```

An I-fetch j operator becomes:

```
L: r2 <- r1 + j       % assume heap address in r1
   r3 <- i-load r2
   ...
```

A heap alloc operator, whose input is a number (the size of the heap object to be allocated) is translated into:

```
L: r1 <- halloc r2       % assume allocation size in r2
   continue
```

Procedure linkage

Suppose we have an Id procedure with two arguments. The P-RISC annotations on the dataflow graphs for procedure call/return and entry/exit are shown on the left and right, respectively, of Figure 16. Recall that FP and FP' are references to the frames of the caller and callee, respectively. As suggested earlier, if N is the entry label of the procedure, then its entry points are labelled N, $N + 1$ and $N + 2$ (and so on, for procedures with more than two arguments). Similarly, if the return label is L, then the entry points for the result and the signal are at L and $L + 1$, respectively.

The return information sent to the callee consists of three items: the return label L in the caller, the frame pointer FP of the caller, and the address of the frame slot rX in the caller's frame where it expects to see the result. We assume that these three items are deposited in the callee's frame at offsets 0, 1 and 2, respectively, which we refer to symbolically as rL, rFP and rrX, respectively. The P-RISC code for the operator at M is:

```
M: join ...             % wait for rFP' and rN
   r1 <- rFP'
   r2 <- L
   fstore r1 r2         % transmit return label
   r1 <- rFP' + 1
   r2 <- current_FP
   fstore r1 r2         % transmit caller's FP
   r1 <- rFP' + 2
   r2 <- #rX
```

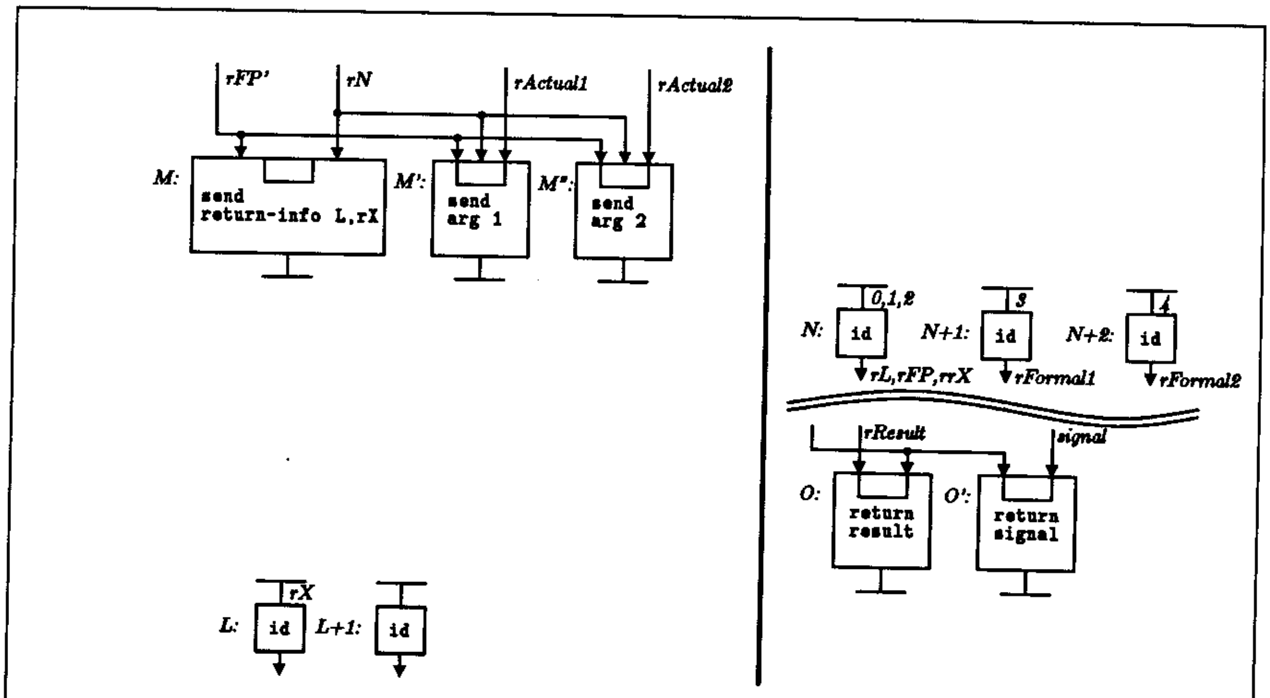

Figure 16: PRISC annotations for procedure call/return (left) and entry/exit (right)

```
        fstore r1 r2              % transmit result frame slot address
        fjump rFP' rN             % initiate callee's trigger thread
```

The notation "#rX" indicates the value "rX" itself (the offset in the frame), rather than the contents of the frame slot named by rX. The P-RISC code for the operator at M' is:

```
    M': join ...                  % wait for rFP', rN and rActual1
        r3 <- rFP' + 3
        fstore r3 rActual1        % transmit first actual parameter
        r3 <- rN + 1
        fjump rFP' r3             % initiate thread that handles it
```

The argument is deposited in the callee's frame at offset 3, which we refer to symbolically as $rFormal1$. The code at M'' is similar, and the generalization for more arguments is obvious. Similarly, the P-RISC code for the operators at O and O' are:

```
    O:  join ...                  % wait for rResult and return-info
        r1 <- rFP + rrX
        fstore r1 rResult         % transmit result
        fjump rFP rL              % initiate result-handling thread

    O': join ...                  % wait for signal and return-info
        r2 <- rL + 1
        fjump rFP r2              % initiate signal-handling thread
```

In the case of the signal (O'), there is no value to return, hence no fstore—it is simply an inter-frame jump to label $L + 1$.

The P-RISC code for the operators at N, $N+1$, ... are simply jumps into the appropriate points of the body of the procedure, *i.e.*, the header of the procedure is simply a jump vector for the various threads of the procedure body.

```
    N  : jump ...          % trigger thread
    N+1: jump ...          % thread for first argument
    N+2: jump ...          % thread for second argument
```

Similarly, the code at L and $L+1$ are simply jumps to the appropriate threads that handle the result and signal.

Top-level driver

A complete Id program will consist of a top-level expression and a collection of top-level definitions. The top-level expression is closed, so that its dataflow graph will have a single input (a trigger) and two outputs (a result and a signal). We drive the top-level graph with the following P-RISC code:

```
START:
    jump ...      % to trigger thread of top-level expression
```

and we arrange for the result and signal threads to jump to `FINISH`:

```
FINISH:
    join ...      % wait for result and signal

    ... print result, clean up, etc. ...
    halt
```

Optimizations

In order to convey the main intuitions of the translation process, our description so far has placed no emphasis on the quality of the P-RISC code that we generate. In this section we outline a variety of code-improvement strategies. Kuszmaul is also investigating several similar optimizations [21].

Faster procedure linkage

The procedure linkage mechanism described above is a general mechanism that supports non-strict, higher-order functions.

The jump table at the top of the procedure allows the caller to derive the labels for all the entry points, given the label of the first entry point. For first-order procedure calls, however, we can elide this jump table completely. Each of the "send" nodes can directly initiate the appropriate threads.

If we have enough strictness information to recognize that certain arguments of a procedure will always be ready before the procedure call, then we do not need to initiate a separate thread for each one. Those arguments could all be sent using `fstore`'s, followed by a single `fjump` (which may fork into separate threads inside the procedure body).

Reducing jumps by better placement of the code

This is an obvious place for improvement. Our translation was described in terms of looking at the nodes of the dataflow graph in isolation, so that each node expanded into P-RISC code that ended in a `jump` to the P-RISC code for another node. By judicious placement of the code, of course, many of these jumps can be removed, allowing the code for one node to "fall through" into the code for its successor.

Removing redundant `fork`'s and `join`'s

In many cases, a little program analysis reveals that certain arcs of the P-RISC graph are redundant. Consider the fragment shown on the left side of Figure 17. The arcs entering the left inputs of the `switch` operators carry the following information: when P-RISC tokens arrive on these arcs, it means that `a` and `b` have been computed and stored in their frame slots. However, the right-hand inputs of the `switch`'es carry the same information—the tokens for those arcs are produced by the "<" operator, which cannot execute until `a` and `b` have been computed. Thus, the left-hand input arcs to the switches carry redundant information. Removing them, we get the graph on the right side of the figure. Thus, we eliminate two `fork`'s and two `join`'s in the resulting P-RISC code.

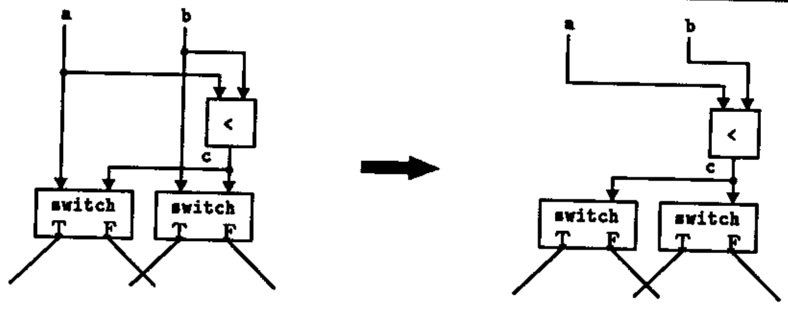

Figure 17: Removal of redundant arcs (and related `fork`'s and `join`'s)

In general, the optimization works as shown in Figure 18. We annotate each arc with the set of all variables (frame slots) in which it is known to be strict. Thus, the left-hand inputs of the `switch`'es are annotated with the sets *(a)* and *(b)*, respectively. The output arc of the "<" operator is annotated its own output variable (`c`) as well as the union of its input annotations, since it is strict in both inputs; thus, it is annotated with the set *(a,b,c)*. Now, at any `join`, if the annotation on one input arc is a subset of the annotation on another, then the former arc can be eliminated. For example, the input arcs of the left switch are annotated with *(a)* and *(a,b,c)*, respectively; therefore, the left input can be removed.

Clearly, the effectiveness of this technique depends on the quality of the strictness information. Instead of the "<" operator, if we had had a call to some unknown binary predicate procedure, then we would have to annotate its output arc with *(c)* only, and we would not have been able to remove any arcs. Strictness analysis [11, 12] is clearly useful in this context.

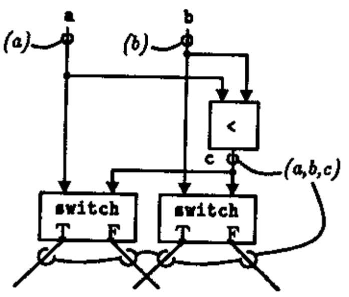

Figure 18: Annotation of arcs with strictness information.

Certain dataflow graph transformations, such as those shown in Figure 19 have the effect of pushing forks downwards and joins upwards. By moving forks towards joins, it improves the likelihood of finding redundant arcs.

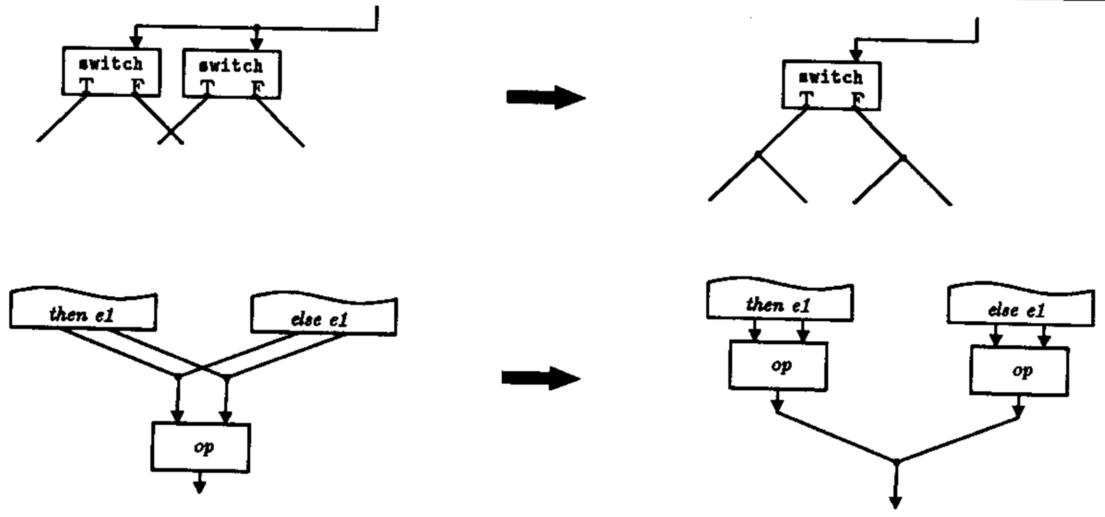

Figure 19: Transformations to bring forks and joins closer

Removing `fork`'s and `join`'s for performance reasons

The previous optimization removed redundant forks and joins. However, it may be beneficial to remove some forks and joins even though they are not redundant.

Consider the dataflow graph on the left side of Figure 20, which computes `(a+b)*(a-b)`. The P-RISC code looks like this:

```
L: fork N
   jump O

M: fork O

N: join ...                 % wait for a and b
```

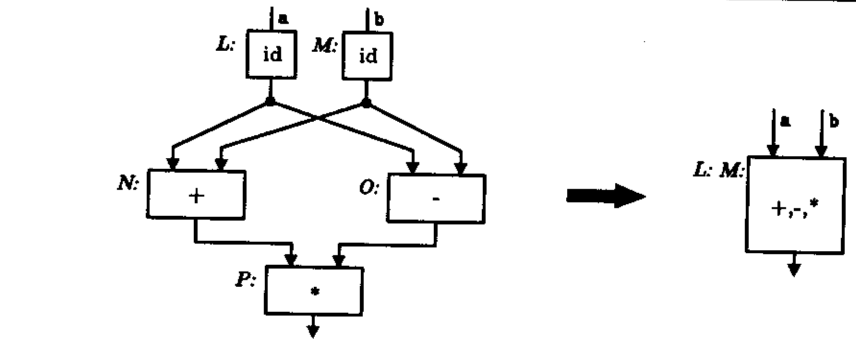

Figure 20: Sequentialization to remover overhead of fork's and join's

```
    r1 <- rA + rB
    jump P

 O: join ...               % wait for a  and b
    r2 <- rA - rB

 P: join ...               % wait for r1  and r2
    r3 <- r1 * r2
    ...
```

However, an alternative would be to lump all three operators together, as depicted on the right side of the figure. The corresponding P-RISC code is:

```
 L:
 M: join ...               % wait for a  and b
    r1 <- rA + rB
    r2 <- rA - rB
    r3 <- r1 * r2
    ...
```

In addition to reducing the total instruction count, this can also improve the behavior of a processor pipeline, since each `join` must be treated like a conditional jump.

What we have given up by using this optimization is the flexibility of allowing the addition and subtraction to be done in any order. In this example, that is clearly worthwhile. However, if we replaced the addition and subtraction with general function calls, it may not be advantageous to force such a sequentialization. The boundary line is not clear, and we need to develop good heuristics.

Another complication is that introducing sequentialization where parallelism is not redundant can introduce deadlock. Suppose we sequentialize two dataflow graphs f and g, *i.e.*, they are normally invoked in parallel, but we force f to go before g. Suppose f contains an `I-fetch` from a location for which the corresponding `I-store` is in g. The sequential version will deadlock because the `I-fetch` will block forever. Traub has described dataflow graph analysis techniques that determine when it is safe to sequentialize two graphs without introducing deadlock [33]. Iannucci presents a heuristic called the Method of Dependence Sets to perform deadlock free partitioning of a dataflow graph [18].

7 Implementation of the P-RISC abstract machine

One can take various approaches to implement the P-RISC abstract machine. We could design and build a special processor that directly executes the P-RISC instruction set. The instruction set is close enough to a conventional RISC instruction set that such an enterprise seems feasible. Madhu Sharma, a member of our group, is investigating this approach [30]. Alternatively, one could map the P-RISC abstract machine onto a stock architecture. In this paper, we will only describe this approach, since such machines will be familiar to most readers. To simplify the exposition, we shall do so in two stages. First, we describe the implementation of the P-RISC abstract machine on a stock uniprocessor. This will establish the basic ideas; then, we describe the implementation on a stock multiprocessor, which is our real target.

Mapping P-RISC to a stock uniprocessor

Our approach is to further translate P-RISC code into native code for a stock uniprocessor. The correspondence between the P-RISC abstract machine to the stock uniprocessor is illustrated in Figure 21. Note that the memory is divided into just two areas—code and heap. All P-RISC data areas (*i.e.*, frames and heap objects) are allocated in a single, common, garbage-collected heap. P-RISC's Token Memory will also be represented on the heap, but not directly, as are frames and heap objects. One of the uniprocessor's registers is used for P-RISC's `FP`, *i.e.*, it holds the frame pointer of the "current token" being executed. We will refer to this register symbolically as `FP`.

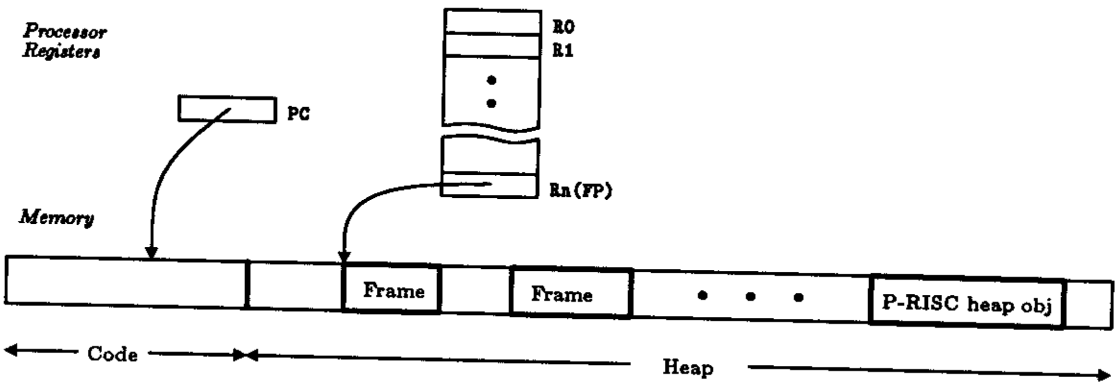

Figure 21: Mapping the P-RISC abstract machine to a stock uniprocessor

We will use an extended version of the C programming language to describe the uniprocessor code. Thus, we can refer to the *j*'th slot of the current frame as `FP[j]`. The only major departure from C will be that we will treat labels as values—we can store labels in a data structure, and later extract them and jump to them. We assume that we have C functions `cons`, `hd`, and `tl` and a C constant `NIL` to construct and manipulate lists.

Our representation of P-RISC's token memory is as follows. Imagine that we sort the token memory according to frame pointers. Imagine that we keep all tokens that have the same frame pointer `FP` in a data structure associated with the frame at `FP`. Clearly, the frame-pointer on each token is now redundant, so we can omit it. Thus, we maintain a list of *instruction pointers* of tokens for a frame, at some reserved offset `ActiveIPs` in the frame. We also maintain a list of all frames that have at least one token in their `ActiveIPs` list. We call this list `ActiveFPs`. The entire structure is shown in Figure 22.

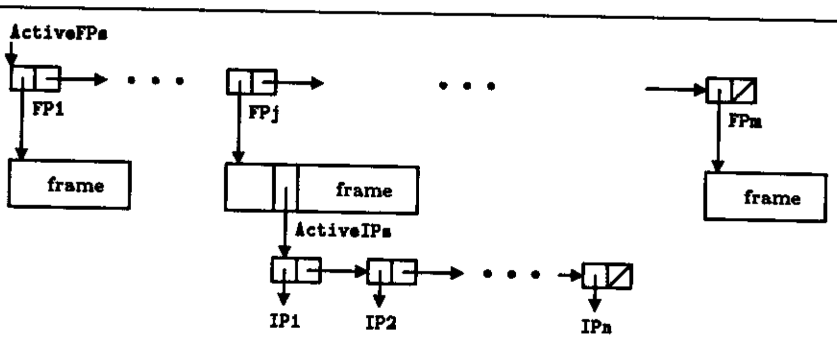

Figure 22: Implementation of a P-RISC frame.

A P-RISC program is a linear sequence of instructions, some of which are labelled, *i.e.*, we label the targets of conditional and unconditional jumps, forks, targets of heap access instructions, *etc.* We will show, for each P-RISC instruction, the corresponding uniprocessor code in our C-like notation.

Arithmetic-Logic

The P-RISC ALU instruction is shown below on the left, and the corresponding uniprocessor code on the right.

```
L: r1 <- r2 + r3       ==>     L: FP[r1] = FP[r2] + FP[r3] ;
M: ...                         M: ...
```

Note that in P-RISC, after the instruction at `L`, the token `(M,FP)` is placed token memory. In the C code, we simply fall through to `M`, *i.e.*, we treat `(M,FP)` as the next token to be processed.

Control

P-RISC unconditional and conditional jumps are simple:

```
L: jump M              ==>     L: goto M ;

N: jzero r1 P          ==>     N: if (FP[r1] == 0) goto P ;
```

A P-RISC `fork` instruction is more interesting. Conceptually, we need to insert a new token in token memory. However, the new token produced by `fork` is always for the current frame. Thus, in our translation, we need to insert just the instruction pointer, *i.e.*, the destination label, onto the list of active `IP`'s in the current frame:

```
L: fork N            ==>     L:  FP[ActiveIPs] = cons(N,FP[ActiveIPs]) ;
M: ...                       M:  ...
```

The other token produced by `fork` in P-RISC is `(FP,M)`. In our C code, we simply fall through to `M`. A P-RISC `join` instruction is translated as follows:

```
L: join r            ==>     L: n = FP[r] - 1 ;
M: ...                          FP[r] = n ;
                                if (n != 0) goto SWITCH_THREAD
                             M: ...
```

The join location in the frame is decremented; if the result is zero, the join succeeds, so we continue by falling through to `M`. If the result is non-zero, the join fails, *i.e.,* we must abandon the current thread, so we jump to `SWITCH_THREAD`:

```
SWITCH_THREAD:
    R1 = FP[ActiveIPS] ;
    if (R1 == NIL) goto SWITCH_FRAME ;
    L = hd(R1) ;
    FP[ActiveIPs] = tl(R1) ;
    goto L ;
```

Here, we remove an instruction pointer from the list of active instruction pointers for this frame, and start executing at that point (again, we remind the reader that this is not legal C notation, which does not allow labels as data values). If the frame has no active instruction pointers, *i.e.,* no waiting threads, we transfer control to `SWITCH_FRAME`:

```
SWITCH_FRAME:
    if (ActiveFPs == NIL) goto IDLE_ERROR ;
    FP = hd(ActiveFPs) ;
    ActiveFPs = tl(ActiveFPs) ;
    goto SWITCH_THREAD ;
```

Here, we change `FP` to point at a new frame that *does* contain at least one waiting instruction pointer, and we start executing at the first waiting instruction pointer in the list: If `ActiveFPs` was empty, then the processor is truly out of work. This could happen, for example, in a program that deadlocks. We jump out to some suitable error handler.

Procedure linkage

An inter-frame store is straightforward:

```
L: fstore r1 r2      ==>     L: FP1 = FP[r1] ;
                                v   = FP[r2] ;
                                *FP1 = v ;
```

An inter-frame jump to `(FP1,IP1)` involves placing the label `IP1` in the list of active instruction pointers for the frame `FP1`.

```
L: fjump r1 r2       ==>     L: IP1 = FP[r1] ;
                                FP1 = FP[r2] ;
                                enqueue_IP(FP1,IP1) ;
                                goto SWITCH_THREAD ;
```

Note that we continue executing in the current frame (by going to `SWITCH_THREAD`). When we enqueue an instruction pointer in the active-IPs list of a frame `FP1`, we must take care of another detail. If the active-IPs list was previously empty and `FP1` is not the current frame, then the frame must have been inactive, and the addition of this instruction pointer now makes it active. Thus, we must now place `FP1` on the list of active frames.

```
enqueue_IP(FP1,IP1)
{
   ips = FP1[ActiveIPs] ;
   FP1[ActiveIPs] = cons (IP1,ips) ;
   if (ips == NIL && FP1 != FP)
       ActiveFPs = cons(FP1,ActiveFPs) ;
}
```

Heap access

We assume the following operations to examine and set the presence bits and contents of a heap location `A`:

```
presence_bits(A)
set_presence_bits(A,state)

contents(A)
set_contents(A,value)
```

where the state of the presence bits is one of the two constants `FULL` and `EMPTY`. When the state is `EMPTY`, we assume that the location contains a list of triples, where each triple consists of a frame pointer, an instruction pointer and a frame offset.

Ordinary loads and stores are straightforward:

```
L: r1 <- load r2        ==>   L: A = FP[r2] ;
                                 FP[r1] = contents(A) ;

M: store r1 r2          ==>   M: A = FP[r2] ;
                                 set_contents(A,FP[r1]) ;
```

A P-RISC `i-load` instruction involves checking if the value is present yet and, if not, queueing the current continuation there and jumping to a new thread:

```
L: r1 <- i-load r2      ==>   L: A = FP[r2] ;
M: ...                           v = contents(A) ;
                                 if (presence_bits(A) != FULL) {
                                     set_contents(A,
                                                  cons(mk_triple(FP,M,r1),
                                                       v)) ;
                                     goto SWITCH_THREAD ;
                                 }
                                 FP[r1] := v
                              M: ...
```

Similarly, a P-RISC `i-store` instruction involves servicing any continuations that may be waiting at that location:

```
L: i-store r1 r2        ==>    L: v = FP[r1] ;
                                  A = FP[r2] ;
                                  if (presence_bits(A)==FULL) goto ERROR ;
                                  triples = contents(A) ;
                                  foreach (FPx,IPx,rx)  in triples {
                                      *(FPx+rx) = v ;
                                      enqueue_IP(FPx,IPx) ;
                                  }
                                  set_presence_bits(A,FULL) ;
                                  set_contents(A,v);
```

Memory management

Both `falloc` and `halloc` are translated into calls to the heap allocator. In addition, `halloc` must initialize all the locations within the object to the empty state, and `falloc` must initialize all join locations with their initial join-counter values.

Mapping P-RISC to a stock multiprocessor

With the basic ideas behind us, we turn to our real target—MIMD machines. As a generic model of currently available MIMD machines, we will assume that our target is a multicomputer consisting of several nodes connected by some network. Each node is a conventional computer, *i.e.*, a processor and a memory, and is capable of sending messages to other nodes.[3] We refer to a node as a "processor-memory element", or PME. The PMEs are numbered 0, 1, 2, ..., p. To send a message from PME i to PME j, the former executes the procedure:

```
send(j,arg1,arg2,...) ;
```

In each case, `arg1` will be a small constant that identifies what kind of message it is: `FSTORE`, `FJUMP`, `LOAD`, `STORE`, `I_LOAD`, or `I_STORE` (we use upper case to distinguish these constants from the similarly-named P-RISC opcodes).

At each PME, we assume that there is a *single* process that executes continuously—this is the "normal" activity of the PME. When a PME receives a message from another PME, we assume that the normal process is interrupted, and the interrupt handler invokes a procedure:

```
receive(args)
{
    ...
}
```

with `args` bound to the arguments that came in the message. The arguments can be accessed by `arg[0]`, `arg[1]`, *etc.* On exit from the `receive` procedure, normal processing is resumed until interrupted by another message, when `receive` is invoked again, and so on. The `receive` interrupt handler and the normal process share data; we will soon discuss the mutual exclusion constraints necessary to ensure consistent access.

[3]Since the mapping from a distributed-memory multiprocessor to a shared-memory multiprocessor is simple and well-known, we do not consider the latter here.

Memory model for P-RISC

The first decision we take is that each P-RISC frame resides entirely within a PME, and that a P-RISC token (FP,IP) is always executed on the PME where the frame FP resides. Thus, frame accesses are always *local* operations (except for the store performed by fstore). The reason for this decision is that we regard a frame as a kind of register set, so that access to frame locations should be like accessing registers.

Different frames may of course reside on different PMEs; indeed, frames are the unit of load balancing. As procedures are invoked, their frames should be allocated on different PMEs in such a manner as to balance computational load. Similarly, each P-RISC heap object resides entirely within a PME, but the various heap objects are distributed across PMEs. Thus, our uniprocessor heap is implemented on a multiprocessor as a distributed heap.

P-RISC frame pointers and P-RISC heap addresses are therefore "global addresses", *i.e.,* a frame pointer FP or a heap address A must be interpreted as a pair consisting of a PME number and an address within that PME. We assume a primitive C macro:

```
pme(A)
```

that takes a global address and returns the PME number where it is located. We also assume a constant currentPME which, on each PME, is bound to the PME number of that PME.

Though the uniprocessor heap is distributed, we replicate the code on all PMEs. Thus, each PME executes a copy of the code using its normal, local instruction fetching mechanism.

P-RISC instruction translation

With this memory model, it should be clear that there is no change in the implementation of P-RISC's ALU and control primitives in going from the uniprocessor to the multiprocessor. The only changes we need to be concerned about are the instructions that may require non-local access, *viz.* the procedure linkage instructions fstore and fjump, and the heap access instructions load, store, i-load and i-store. We show the translations below; in each case, the translation is a small generalization of the uniprocessor version. A key feature is that all non-local accesses are implemented as split-phase transactions, *i.e.,* separate request and response messages.

```
L: fstore r1 r2        ==>    L: FP1 = FP[r1] ;
                                 v   = FP[r2] ;
                                 p   = pme(FP1) ;
                                 if (p != currentPME)
                                     send(p,FSTORE,FP1,v);
                                 else
                                     *FP1 = v ;
```

The fjump translation is unchanged, but we redefine the enqueue_IP(FP1,IP1) procedure that it calls. If FP1 is a remote frame, we send an FJUMP request to the remote PME. Otherwise, as usual, we enqueue the IP on the local frame and activate the frame if it was not previously active.

```
                              enqueue_IP(FP1,IP1)
                              {
                                 p = pme(FP1) ;
                                 if (p != currentPME)
                                      send(p,FJUMP,FP1,IP1) ;
                                 else {
                                      ips = FP1[ActiveIPs] ;
                                      FP1[ActiveIPs] = cons (IP1,ips) ;
                                      if (ips == NIL && FP1 != FP)
                                           ActiveFPs = cons(FP1,ActiveFPs) ;
                                 }
                              }
```

In a `load`, if the address is remote, we *always* switch to another thread after sending off the request. Thus, the message is the first phase of a split-phase transaction, and the current PME is *never* made to wait for a response from a remote PME.

```
L: r1 <- load r2        ==>   L: A = FP[r2] ;
M: ...                           p = pme(A) ;
                                 if (p != currentPME) {
                                      send(p,LOAD,A,FP,M,r1) ;
                                      goto SWITCH_THREAD ;
                                 }
                                 FP[r1] = contents(A) ;
                              M: ...
```

Similarly, a `store` either stores a value locally, or sends a message for the value to be stored remotely.

```
N: store r1 r2          ==>   N: A = FP[r2] ;
                                 v = FP[r1] ;
                                 p = pme(A) ;
                                 if (p != currentPME)
                                      send(p,STORE,A,v) ;
                                 else
                                      set_contents(A,FP[r1]) ;
```

The P-RISC `i-load` instruction and `i-store` instruction are similarly translated:

```
L: r1 <- i-load r2      ==>   L: A = FP[r2] ;
M: ...                           v = contents(A) ;
                                 p = pme(A) ;
                                 if (p != currentPME) {
                                      send(p,I_LOAD,A,FP,M,r1);
                                      goto SWITCH_THREAD ;
                                 }
                                 if (presence_bits(A) != FULL) {
                                      set_contents(A,
                                                 cons(mk_triple(FP,M,r1),
                                                      v)) ;
                                      goto SWITCH_THREAD ;
                                 }
                                 FP[r1] := v
                              M: ...

L: i-store r1 r2        ==>   L: v = FP[r1] ;
                                 A = FP[r2] ;
```

```
p = pme(A) ;
if (p != currentPME)
    send(p,I_STORE,A,v) ;
else {
    if (presence_bits(A)==FULL) goto ERROR ;
    triples = contents(A) ;
    foreach (FPx,IPx,rx)  in triples {
        *(FPx+rx) = v ;
        enqueue_IP(FPx,IPx) ;
    }
    set_presence_bits(A,FULL) ;
    set_contents(A,v);
}
```

Receiving messages

So far, we have seen how various P-RISC instructions result in sending messages to remote PMEs. On each PME, we assume that when a message arrives, it creates an interrupt and that the interrupt handler calls the following procedure. The procedure dispatches to various message-handlers based on the first argument, which specifies what kind of message it is.

```
receive(args)
{
    switch (args[0]) {
      case FSTORE:   do_fstore(args[1],args[2]) ;                      break ;
      case FJUMP:    do_fjump(args[1],args[2]) ;                       break ;
      case LOAD:     do_load(args[1],args[2],args[3],args[4]);         break ;
      case STORE:    do_store(args[1],args[2]) ;                       break ;
      case I_LOAD:   do_i_load(args[1],args[2],args[3],args[4]);       break ;
      case I_STORE:  do_i_store(args[1],args[2]) ;                     break ;
    }
}
```

We describe the various message-handlers below, but first we address the issue of atomicity. Many of the messsage handlers both read and write locations that are also accessible to the normal process of the PME (such as heap and frame locations, and PME data structures such as ActiveFPs). To ensure that these shared locations are read and updated consistently, message interrupts should only be allowed to occur at certain "safe" points in the normal process. Examples of such safe points are: between "logical" P-RISC instructions, or at the SWITCH_THREAD label. Of course, if messages arrive more frequently than this, the interrupt handler could merely buffer them in a queue, to be handled later when the normal process reaches a safe point.

FSTORE messages just store into a frame location.

```
do_fstore(FP1,v)
{
    *FP1 = v ;
}
```

FJUMP messages enqueue the incoming FP and IP onto the appropriate frame.

```
do_fjump(FP1,IP1)
```

```
    {
        ips = FP1[ActiveIPs] ;
        FP1[ActiveIPs] = cons (IP1,ips) ;
        if (ips == NIL && FP1 != FP)
            ActiveFPs = cons(FP1,ActiveFPs) ;
    }
```

If the frame was not active, and it is not the current frame in this processor, it is added to the `ActiveFPs` list.

`LOAD` messages read a heap location and send the value back, whereas `STORE` messages just store a value into a heap location:

```
    do_load(A,FP1,IP1,r)
    {
        p = pme(FP1) ;
        v  = contents(A) ;
        send(p,FSTORE,FP1+r,v);
        send(p,FJUMP, FP1  ,IP1) ;
    }

    do_store(A,v)
    {
        set_contents(A,v) ;
    }
```

`I_LOAD` messages test the presence bits of a heap location. If empty, the return-continuation information is enqueued; otherwise the value in the location is sent back:

```
    do_i_load(A,FP1,IP1,r)
    {
        v  = contents(A) ;
        if (presence_bits(A) != FULL)
            set_contents(A,
                         cons(mk_triple(FP,M,r1),
                              v)) ;
        else {
            p = pme(FP1) ;
            send(p,FSTORE,FP1+r,v);
            send(p,FJUMP, FP1,  IP1) ;
        }
    }
```

`I_STORE` messages store a value in a heap location. If there are any waiting readers, the value is sent to all of them. Waiting readers may be local or remote.

```
    do_i_store(A,v)
    {
        if (presence_bits(A)==FULL) goto ERROR ;
        triples = contents(A) ;
        foreach (FPx,IPx,rx)  in triples {
            p = pme(FPx) ;
            if (p != currentPME) {
                send(p,FSTORE,FPx+rx,v) ;
                send(p,FJUMP, FPx,   IPx) ;
            }
            else {
                *(FPx+rx) = v ;
                enqueue_IP(FPx,IPx) ;
            }
        }
        set_presence_bits(A,FULL) ;
        set_contents(A,v);
    }
```

Termination

In the uniprocessor version, it was an error for SWITCH_FRAME to find no remaining active frames—it was an indication of deadlock. In the multiprocessor, this is not the case. Even though at some point there may be no active frames in a PME, it may later receive a message that activates some frame. Thus, we change SWITCH_FRAME to wait for work:

```
SWITCH_FRAME:
    if (ActiveFPs == NIL) goto SWITCH_FRAME ;
    FP = hd(ActiveFPs) ;
    ActiveFPs = tl(ActiveFPs) ;
    goto SWITCH_THREAD ;
```

Thus, we busy-wait at SWITCH_FRAME, but there is really nothing else that the processor can do anyway. When a message finally arrives, the interrupt handler will put a frame onto the ActiveFPs list, and the main process will fall out of the loop.

The main PME, when it enters the code at FINISH, can send FJUMP messages to all the other PME's containing an arbitrary frame pointer and instruction pointer SUB_FINISH. This simply forces them all out of the busy-wait loop into termination code:

```
SUB_FINISH:
    ... clean up, terminate ...
```

However, this means that if there really is a deadlock, then all PMEs will be spinning at SWITCH_FRAME. Thus, we will need some kind of distributed deadlock detection algorithm running concurrently with the main program.

Memory management and load balancing

We have deliberately not shown the translation of the P-RISC falloc (frame allocation) and halloc (heap allocation) instructions because these can be quite complex. First of all, they are distributed allocators, and some form of heirarchical allocation method is necessary to avoid bottlenecks. For example, each PME has a local allocator which is consulted first, and when the local allocator runs out of space, it consults a global allocator for another large chunk of space, and so on. Each allocator must spread its allocations across PMEs in order that frames and heap objects are distributed evenly amongst the PMEs. Finally, a garbage collector is necessary for reclaiming unreachable objects.

P-RISC, as such, provides no special advantages or disadvantages in coding these distributed storage management algorithms beyond those already cited for the main program.

For example, calls to the allocator can be implemented as split-phase operations, so that the allocator code can be run concurrently with normal processing, and PMEs do not have to block when one thread calls an allocator.

Discussion

The data-driven mechanism

Having laid out the entire framework, it is now worth reemphasizing that in our compiling/runtime/architecture system, we have arranged for *all* non-local communications to be non-blocking, split-phase transactions. In other words, when PME A needs information from PME B, whether *via* a procedure call or a heap access, PME A never has to wait for PME B's response—it can always do some other work in the interim (assuming there is enough parallelism in the program). By going to very fine-grained threads, we increase the likelihood that there will indeed be adequate parallelism in the program to keep all PME's busy. All non-local communications carry enough continuation-information such that the target PME does not need any special mechanism to associate messages with waiting processes. We firmly believe that no matter what compilation method one may use, this runtime model—numerous fine-grained threads, non-blocking split-phase communications, and direct continuation information in messages—is essential in order to achieve high utilizations in massively parallel MIMD machines.

Ordering, scheduling, and locality issues

In Section 6, after describing the translation from dataflow graphs to P-RISC code, we mentioned some optimizations that impose an ordering on the operations of the graph (see [32] for a much more detailed discussion of how such ordering may be imposed).

In this section, the reader may have noticed that, while the P-RISC abstract machine did not specify the order in which tokens in the token memory are to be executed, we made a static ordering decision in the translation to C code. For example, after a P-RISC ALU instruction at `(FP,IP)`, we always execute the P-RISC instruction at `(FP,IP+1)` by just falling through into the C code for it. Similarly, for a `fork` instruction, we have statically chosen to continue executing along one tine of the fork (at `IP+1`) while enqueuing the other.

Another ordering decision that we have taken is dynamic, *i.e.,* it is a scheduling policy. In the code at `SWITCH_THREAD`, we have chosen to continue executing within the current frame as long as there are active instruction pointers for that frame, *i.e.,* we go to `SWITCH_FRAME` only when `ActiveIPs` becomes empty.

What is the rationale for all these ordering decisions? The answer is that for each target architecture, there is typically some notion of *locality* which may be exploited to achieve better performance.

One such notion of locality concerns threads and registers. In our description so far, all values are communicated from one instruction to another *via* frame or heap locations which are persistent across SWITCH_THREAD and SWITCH_FRAME, and are thus assumed to be in main memory. However, for an unbroken thread, it is possible for instructions to communicate values *via* processor registers, which are typically much faster than main memory. We assume that processor registers are volatile, *i.e.,* they do not survive across SWITCH_THREAD, because threads are too short to afford saving and restoring registers on thread switches. Thus, it is advantageous to lengthen threads as much as possible (without losing useful parallelism) in order that more values can be communicated down threads via registers. Another reason to lengthen threads is that it allows the processor's instruction pre-fetch mechanism to do a better job.

The decision to continue executing within a single frame as long as it has active instruction pointers has other advantages. The processor's cache is likely to exhibit better behavior since data accesses remain within a single frame, and instruction accesses remain within a single procedure for a longer period of time. Many recent proposals for multi-threaded architectures [14, 30, 1] assume similar locality within a frame. Further, some of the frame locations could be maintained in the processor registers, to be saved only when we jump to SWITCH_FRAME.

In describing the translations, we did not pay any attention to the variations in the size of frames for different procedures. For very fast allocation and deallocation, on the other hand, particularly in a parallel machine, it may be preferable to use fixed-size frames for all procedures. This can be achieved by a combination of inlining of small procedures and splitting of large procedures.

Our implementation does not use general message passing. It uses a fixed repertoire of message types, with each type having a fixed size. This fact can be used to optimize the message passing mechanism to minimize its latency.

8 Conclusion

In [4], Arvind and Iannucci analyzed massively parallel MIMD machines to show that for general-purpose computation on such machines, programs must be broken into a very large number of small threads (many more than the number of processors); that all non-local communications should be non-blocking, split-phase transactions; and, that there should be a fast thread switching mechanism. We addressed the first requirement by using a non-strict functional programming language with a lenient evaluation strategy, and the remaining requirements by compilation *via* intermediate forms that retain fine-grained parallelism and support split-phase transactions.

There are three projects that are closely related to the work described here. In our group, Bradley Kuszmaul is working on a compiler for Id for sequential uniprocessors [21]. While he starts with a P-RISC-like intermediate form, his subsequent transformations are not constrained by the requirements of MIMD machines. At Berkeley, David Culler and his group are implementing a new back end for the Id compiler which we believe is substantially similar to the approach described here. At IBM Research, Robert Iannucci is building a hybrid von Neumann-dataflow machine in the EMPIRE project, and K. Ekanadham is building an Id compiler for it by retargeting the MIT Id compiler. They are designing an intermediate language called "kudos" which, like P-RISC, also has fine-grained threads and synchronization.

Most of the techniques described here have broader applicability than functional languages. The translation from dataflow graphs to P-RISC code and to machine code is independent of the method by which those dataflow graphs were originally produced. In particular, dataflow graphs are not unique to functional languages. Ballance, Maccabe and Ottenstein in New Mexico [7], and Beck and Pingali at Cornell [9] have made great progress in showing how to translate imperative languages, including FORTRAN, into dataflow graphs. In our own group, Barth and Nikhil [8] have been experimenting with a programming construct called "managers" for fine-grained, parallel access to shared, updateable objects, and these are also translated into dataflow graphs.

The P-RISC back end described here not yet been constructed. Once it is ready, we have much work ahead of us in analyzing the code produced, improving the compiler's optimizations, and, most importantly, investigating the resource-management issues in frame and heap management.

Acknowledgements

We would like to thank the members of the Computation Structures Group in MIT's Laboratory for Computer Science and Professor Arvind in particular for providing such an invigorating environment which encompasses experts on functional languages, compilation technology, and parallel architectures. Madhu Sharma, Harry Hochheiser and especially Ken Traub provided detailed comments on a draft of this paper.

Funding for the group is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts (LCS General:) N00014-83-K-1025 and (MEF:) N00014-84-K-0099.

We would like to thank Dr. Mario Furnari for inviting us to participate in the Workshop on Massive Parallelism in Amalfi, October 1989, where we presented a summary of this work.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proc. 17th Annual Intl. Symp. on Computer Architecture, Seattle, Washington, U.S.A.*, pages 104–114, May 28-31 1990.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, Reading, Massachusetts, USA, 1986.

[3] A. Appel and D. B. MacQueen. A Standard ML Compiler. In *Proceedings of the Conference on Functional Programming and Computer Architecture, Portland, Oregon,* September 1987. Springer-Verlag LNCS 274.

[4] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, Springer-Verlag LNCS 295*, June 25-29 1987.

[5] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[6] L. Augustsson and T. Johnsson. Parallel Graph Reduction with the $< \nu, G >$-machine. Technical Report Draft, Programming Methodology Group, Department of Computer Science, Chalmers University of Technology, S-412 96 Goteborg, Sweden, March 10 1989.

[7] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A Representation Supporting Control-, Data- and Demand-Driven Interpretation of Imperative Languages. In *Proc. SIGPLAN '90 Conf. on Programming Language Design and Implementation*, July 1990.

[8] P. S. Barth and R. S. Nikhil. Supporting State-Sensitive Computation in a Dataflow System. Technical Report CSG Memo 294, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, March 1989.

[9] M. Beck and K. K. Pingali. From Control Flow to Dataflow. Technical Report TR89-1050, Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, USA, October 1989.

[10] G. L. Burn. Overview of a Parallel Reduction Machine Project II. Technical Report 126, GEC Hirst Research Centre, East Lane, Wembley, Middlesex HA9 7PP, UK, March 1989.

[11] G. L. Burn, C. Hankin, and S. Abramsky. The Theory and Practice of Strictness Analysis for Higher Order Functions. Technical Report Research Report DoC 85/6, Department of Computing, Imperial College of Science and Technology, April 1985.

[12] C. Clack and S. L. Peyton Jones. Strictness Analysis - A Practical Approach. In *Proc. Functional Programming Languages and Computer Architecture, Nancy, France, September 1985*, pages 35–49, September 1985. Springer-Verlag LNCS 201.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[14] R. H. Halstead Jr. and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the IEEE 15th. Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, June 1988.

[15] J. E. Hicks Jr. *Compiler Directed Storage Reclamation by Object Lifetime Analysis.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, May 1991 (expected).

[16] P. Hudak and D. Kranz. A Combinator-based Compiler for a Functional Language. In *Proc. 11th. Annual ACM Symp. on Principles of Programming Languages, Salt Lake City, Utah, USA*, pages 122–132, January 15-18 1984.

[17] P. Hudak and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, Apr. 1990.

[18] R. A. Iannucci. *Parallel Machines: Parallel Machine Languages The Emergence of Hybrid Dataflow Computer Architectures.* Kluwer Academic Publishers, Boston, MA., U.S.A., April 1990. ISBN 0-7923-9101-2.

[19] D. Kranz. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Department of Computer Science, Yale University, 1988.

[20] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[21] B. C. Kuszmaul. *Compiling Data-Flow Programs for Control-Flow Computers.* PhD thesis, Massachusetts Institute of Technology, Computation Structures Group, MIT Labratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, U.S.A., 1991 (expected).

[22] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[23] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1990.

[24] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th. Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 262–272, May 29-31 1989. Also: CSG Memo 292, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

[25] R. S. Nikhil and Arvind. *Programming in Id: a parallel programming language.* 1990. (Book in preparation).

[26] G. M. Papadopoulos and D. E. Culler. Monsoon: An Explicit Token Store Architecture. In *Proc. 17th. Intl. Symp. on Computer Architecture, Seattle, WA*, May 1990.

[27] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[28] S. L. Peyton Jones. The Spineless Tagless G-machine. In *Proceedings of the Workshop on Implementation of Lazy Functional Languages, Aspenas, Sweden*, September 5-8 1988.

[29] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP – A High Performance Architecture for Parallel Graph Reduction. In *Proceedings of the 3rd. International Conference on Functional Programming and Computer Architecture, Portland, Oregon*, September 1987.

[30] M. Sharma and R. S. Nikhil. PRISC-1: A Multi-threaded RISC Architecture. Technical Report (unpublished), MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, November 1989.

[31] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

[32] K. R. Traub. Sequential Implementation of Lenient Programming Languages. Technical Report TR-417, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1988. Ph.D. thesis.

[33] K. R. Traub. Compilation as Partitioning: A New Approach to Compiling Non-Strict Functional Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, London*, September 1989.

[34] D. A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *Proc. Functional Programming Languages and Computer Architecture, Nancy, France (Springer-Verlag LNCS 201)*, pages 1–16, September 1985.