Taming the IXP Network Processor *

Lal George
Network Speed Technologies, Inc
lg@network-speed.com

Matthias Blume
Toyota Technological Institute at Chicago
blume@tti-c.org

ABSTRACT

We compile Nova, a new language designed for writing network processing applications, using a back end based on integer-linear programming (ILP) for register allocation, optimal bank assignment, and spills. The compiler's optimizer employs CPS as its intermediate representation; some of the invariants that this IR guarantees are essential for the formulation of a practical ILP model.

Appel and George used a similar ILP-based technique for the IA32 to decide which variables reside in registers but deferred the actual assignment of colors to a later phase. We demonstrate how to carry over their idea to an architecture with many more banks, register aggregates, variables with multiple simultaneous register assignments, and, very importantly, one where bank- and register-assignment cannot be done in isolation from each other. Our approach performs well in practise—without causing an explosion in size or solve time of the generated integer linear programs.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*

General Terms

Algorithms, Performance, Languages.

Keywords

network processors, Intel IXA, integer linear programming, register allocation, bank assignment, programming languages, code generation.

1. INTRODUCTION

Network processors are designed to meet the demands of next generation networks: cost, scalability, and performance of packet-manipulation applications. To achieve very high execution speed when processing data at gigabit line rates, network processors such as the Intel IXP employ fairly unusual designs which make it hard to write programs for them. Our work is an attempt to address this problem. We have focused on the IXP1200, but all of the ideas carry over to newer generations of the architecture.

The IXP1200 consists of a StrongARM core and six micro-engines with hardware supported multi-threading. Figure 1 shows the basic architecture as seen from the vantage point of a single micro-engine thread. There are six register banks: two general purpose banks (**A** and **B**); two banks forming the interface to external SRAM memory (**L** and **S**); and two as the interface to external SDRAM memory (**LD** and **SD**). The **L** and **LD** transfer banks are the destinations for all memory loads, **S** and **SD** the sources of all stores.

Input to the ALU can come from **L**, **LD**, **A**, or **B**, but each of **A**, **B**, and **L** ∪ **LD** can supply at most one operand. Results from the ALU can go to **A**, **B**, **S**, or **SD**. There is no direct path from any register in a transfer bank to another register in the *same* transfer bank. Not shown in the figure is an on-chip scratch memory **M**, also accessed via **L** and **LD**.

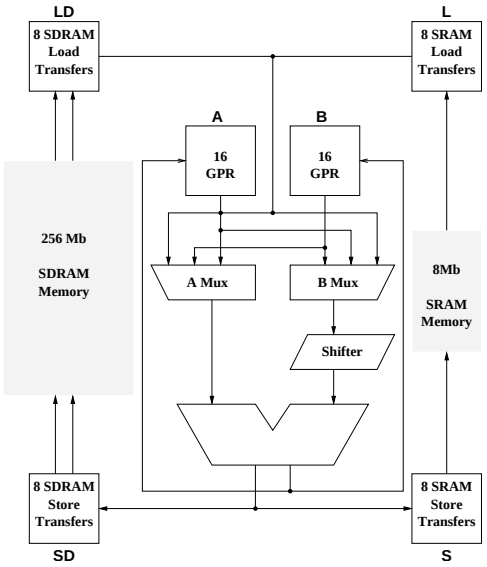


Figure 1: Micro-engine architecture

1.1 IXP programming issues

To the compiler, the IXP hardware presents a combination of several difficult problems for which there are no good published heuristics. As a result, the state-of-the-art in programming the IXP is still (a very quirky) assembly. A high-level programming lan-

*This works was done at Lucent Technologies, Bell Labs.

PLDI'03, June 9–11, 2003, San Diego, California, USA.

guage for the IXP and its compiler must address:

Few registers: Because of the penalty for memory accesses, lack of data caches, and real-time constraints, spilling (not to mention the use of a stack) is nearly intolerable.

Bank assignment: The IXP has many different register banks with quite different characteristics. Which program variable should be allocated to which bank and when?

Register aggregates: Transactions to memory are performed in sets of adjacent registers called *aggregates*. Several aggregate sizes anywhere in the range $1 \ldots 8$ occur in most programs. Where should an aggregate be allocated within a bank? If the bank is fragmented, which variables should be moved out to accommodate the aggregate, where should the evicted values go, and when should this happen?

Limited data paths: After a variable has been moved to **S** or **SD**, it cannot be moved back to another bank without going through memory. If its value is required elsewhere in the program, then it should have been duplicated before being moved. When and how should such duplication occur?

Data structures and alignment: Access to SDRAM memory is restricted to 8-byte boundaries and access to SRAM to 4-byte boundaries. Real-world packet data does not respect these alignment requirements. How can one effectively deal with misalignment in conjunction with header field extraction?

Fine control: How should one provide the necessary knobs to access specialized hardware registers for I/O and concurrency control in a high level language?

1.2 Our Approach and Contributions

The Nova language and its compiler address all of these issues with good results:

Optimal bank assignment: Our formulation of integer-linear programming (ILP) generates an optimal bank assignment including spill considerations.

Allocation of aggregates: Allocation of aggregates strongly interacts with bank assignment and is difficult to solve heuristically. Therefore, we use ILP to solve the two problems together.

Static single use: Our compiler makes use of a *static single use* property enforced for certain variables, enabling the register allocator to place these variables into multiple registers at the same time when doing so is beneficial.

Typed language: Nova's static type system is stratified into two layers: *types* and *layouts*. Record- and tuple types describe collections of word-size data that typically would be stored in collections of registers. Other types are related to the management of Nova's control structures. The typing rules guarantee that no memory allocation (stack or heap) is required for implementing control.

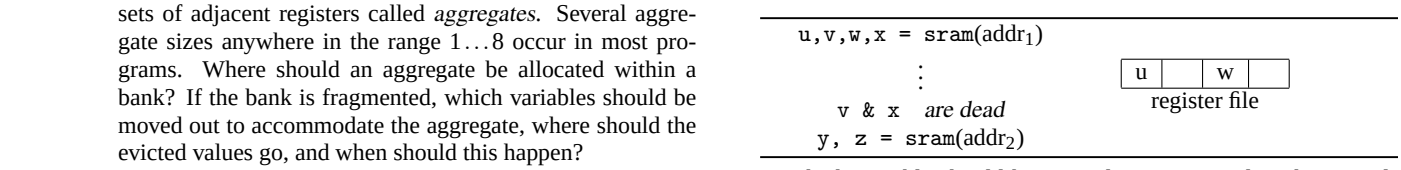
Layouts: Layouts elegantly deal with alignment issues and bit-level data access, resulting in efficient and easy-to-maintain code.

Practical demonstration: Experimentation with our prototype provides evidence that the techniques are successful in compiling real-life programs—with compile times short enough to accommodate an edit-compile-debug cycle.

2. RELATED WORK

2.1 Code Generation

Imagine a "mini-IXP" where the transfer banks have four registers, and consider a program that first fills the **S** bank with values `u`, `v`, `w`, and `x`. Along some control path starting at this instruction, the variables `v` and `x` might go dead, leaving the register file with holes. Later on, the program might require sufficient contiguous space for allocating `y` and `z`:



Which variable should be evicted to accommodate the new allocation? Either `u` or `w` would do, as sufficient space would then be made available. But the best choice might also depend on lifetime, location, and usage constraints of the other variables in the program. The literature is filled with both heuristic and ILP-based techniques for similar problems. However, none of them apply directly to the IXP, nor can they easily be adapted.

For example, past work on handling floating point register pairs or overlapping registers (such as `AL`, `AX`, `EAX` on the IA32) makes the assumption that something to be allocated to the larger register will always stay together. This results in a simpler picture since individual smaller-sized pieces will have identical live ranges and never get considered in isolation from each other [19, 21].

In their work on compiling for irregular architectures, Kong and Wilken [19] as well as Scholz and Eckstein [21] insert register-register copies at strategic points to start a new live range. This gives their algorithm more freedom in the choice of register banks or memory. Their optimization technology is ultimately responsible for determining if the copies are necessary. Given that there are six register banks on the IXP, it is not clear where such copies should be inserted and what they would accomplish. In contrast, we do not insert copies into our input program, but our ILP formulation is free to insert an inter-bank move at any program point as necessary.

Tallman and Gupta [24] describe a bit-width aware register allocation algorithm where static analysis is used to compute the variations in the bit-width over the lifetime of a variable. This information is used to construct an interference graph that aids in packing several variables into a single register. The allocation of aggregates to transfer banks can be seen as a similar packing problem (at register-granularity). Unfortunately, the analogy does not carry very far since IXP aggregates do not have to be stored in adjacent registers all of the time. There are only a few specific program points where aggregation matters; at all other points it does not have to be taken into account (at least not directly). Not even the live ranges of individual members of any given aggregate will necessarily coincide.

Fu and Wilken [15] investigated the problem of reducing the number of variables and constraints in an ILP model for the IA32 without compromising optimality with the goal of achieving faster solve times. As described later (Sections 8 and 9), we had to deal with similar problems, arriving at similar solutions.

Dealing with multiple register files in clustered VLIW architectures dates back to the Bulldog compiler [12]. The problem in clustered architectures is that of keeping related registers together. On the IXP we face the dual problem of keeping the operands of a single instruction in different banks, taking the architecture's limited data paths between banks into account.

The register allocation and bank assignment on the IXP is difficult enough that we do not consider the integration of scheduling, or memory bank assignment. This is a shortcoming that will have to be addressed in future work. Since memory latencies differ from bank to bank, a solution to this problem will have to take the multi-threaded nature of IXP applications into account.

Let us come back to our mini-IXP with transfer banks of size four. Consider a following program that contains two SRAM store instructions with a common operand `x`:

```
sram(addr1) = u, v, x, w
  ⋮
sram(addr2) = a, x, b, c
```

The different positions of `x` in the list of operands creates conflicting constraints for its allocation to a register in the transfer bank. We could address this problem by making copies of `x`, using a different copy in each of the conflicting instructions. Since making a physical copy increases register pressure, optimal placement of the copy instruction is intertwined with optimal register- and bank assignment itself. Not always are all copies required. Which ones are, however, depends on other decisions of the register allocator. For example, suppose the last use of `x` is in an instruction like:

```
sram(addr3) = y, x, z
```

This use of `x` is compatible with one of the other two uses (but not with both), so an existing copy of `x` might be able to do double-duty here. But not before register allocation is done do we know which one, if any. The traditional approach is to make sufficiently many copies when `x` is defined and to rely on subsequent coalescing to eliminate the ones that are unneeded [16]. The idea of *cloning* (see Section 10) makes it possible for the same technique to work in an ILP-based allocator.

2.2 Programming Packet Manipulation

From the programmer's point of view, one of the most tedious and error-prone aspects of packet manipulation is that of header field extraction. Depending on sizes and offsets, a field might be aligned with the start or the end of a word, it might reside inside a word, or it could straddle a word boundary.

To extract a field requires a different sequence of instructions in each case, and the slightest change to one of the sizes has the potential of changing the situation for many fields, making it difficult to maintain or enhance hand-crafted machine code. The few existing compilers for higher-level languages on the IXP (such as Intel-C) do not provide the language support necessary to improve the situation.

The next section presents an overview of the Nova programming language with emphasis on the salient design choices, and the semantics of layout and overlay specifications. Section 4 gives an overview of the CPS-based front end of our prototype implementation. The bulk of the paper (Sections 5 through 10) presents our formulation of the ILP optimization problem, and lastly we finish with performance results on stock hardware and conclusions.

3. THE PROGRAMMING LANGUAGE

Nova is a modern, lexically-scoped, strict, statically typed, left-to-right order, call by-value programming language. It has many of the familiar control constructs including functional abstraction and exceptions, but can be compiled to a FORTRAN-like runtime model.

Nova provides value aggregation through record- and tuple-types as well as arrow- and exception-types for describing arguments that are functions or exceptions. There is no polymorphism.

3.1 Inexpensive to Implement

Compiled Nova code must be able to fit within one or two thousand words of IXP microengine instruction cache. The code will typically implement the fast path of some network processing application and must be as efficient as possible.

Our language design aims at helping in this regard by avoiding constructs that could become expensive. In particular, unless register allocation requires spills (which is very rare), memory allocation and memory access are never implicit but always syntactically apparent. This means that a number of popular features have been intentionally left out:

no recursive types Nova does not have support for recursive algebraic types such as lists or trees because, in general, values of these types would have to be memory-allocated.

no stack Nova functions can be mutually recursive, but all recursive calls are restricted to be in tail-call position. Furthermore, the absence of recursive types prohibits self-application, making it impossible to define a fixpoint combinator that would bring unrestricted recursion through the back-door.

no memory-allocated closures Nova functions can be nested so that free occurrences of variables in an inner function refer to their corresponding definitions in the outer scope. But no two function values created by instantiating a single definition can ever be alive simultaneously in Nova programs, so closures do not have to be memory-allocated.

flattening of records Our language has records and tuples with fields of arbitrary types. Records are finite collections of labeled values, written by enclosing them in square brackets (e.g., `[x=4, y=3]`), while tuples are sequences of values, written by enclosing them in parentheses (e.g., `(4,3)`). The grouping of conceptually related values into larger values is a useful organizing tool for the programmer, but it does not have to be reflected in the runtime model. Our compiler handles tuples and records by compile-time bookkeeping; only leaf fields have a runtime counterpart, and each of them is treated as an independent variable by the register allocator.

3.2 Layouts

The analysis of incoming and the construction of outgoing network packets is at the heart of network processing. Therefore, Nova programs perform many bit- or byte-level operations, i.e., masking and shifting.

The instruction set of the IXP microengine provides adequate support for dealing with these situations, but hand-crafting the necessary code is tedious and error-prone. Moveover, even a small update such as the insertion of a new field at the beginning of a header typically means that any code using these low-level operations must be rewritten from scratch.

The Nova language automates the task of generating shift and mask operations by providing a sublanguage for defining *layouts* that statically describe the arrangement of bitfields within a byte stream. For every layout *l* Nova defines two types: *packed*(*l*) and *unpacked*(*l*). The former is a sufficiently long word tuple type describing raw, packed data with all bits in their correct positions. The latter is a record type whose structure follows the structure of

l's definition where all bitfields have been spread out, each into its own record component of type *word*.

For example, the layout definition of the IPv6 header lists its fields and makes use of another layout called *ipv6_address*:

```
layout ipv6_address =
  { a1 : 32, a2 : 32, a3 : 32, a4 : 32 };

layout ipv6_header = {
  version        : 4,
  priority       : 4,
  flow_label     : 24,
  payload_length : 16,
  next_header    : 8,
  hop_limit      : 8,
  src_address    : ipv6_address,
  dst_address    : ipv6_address
};
```

Given this definition, the type name *packed*(*ipv6_header*) is a synonym for *word[10]* and we have the following type equalities:

```
type unpacked(ipv6_address) =
  [ a1: word, a2: word, a3: word, a4: word ]


type unpacked(ipv6_header) =
  [ version        : word,
    priority       : word,
    flow_label     : word,
    payload_length : word,
    next_header    : word,
    hop_limit      : word,
    src_address    : unpacked(ipv6_address),
    dst_address    : unpacked(ipv6_address) ]
```

On the operational side, *packed*(*l*) and *unpacked*(*l*) are connected by operations **unpack**[*l*](x) and **pack**[*l*](x). The **unpack**[*l*] operator maps a value of type *packed*(*l*) to the corresponding value of type *unpacked*(*l*). For example, one could write something like:[1]

```
let pdata : packed(ipv6_header) = ...
let udata = unpack[ipv6_header](pdata);
if (udata.version == 6 && udata.hop_limit > 0)
  ...
else ...
```

Although formally every bitfield mentioned in *l* gets extracted by **unpack**[*l*](x), no actual machine instructions will be generated for those fields that are ignored by the rest of the program.

A formal definition for **pack**[*l*], which acts as the inverse of **unpack**[*l*], is complicated by the possibility of *overlays* within layouts. Overlays consist of two or more alternative sub-layouts, each covering the same bit range. Unpacking generates all bitfields in a given layout, including every possible alternative of each of its overlays, but packing takes input corresponding to precisely one alternative of each overlay.

As an example, consider the aforementioned IPv6 address layout. In most programs it makes sense to consider `version` and `priority` fields together, forming a larger 8-bit field that is cheaper to extract. A revised version of our layout could provide the two competing views side-by-side in the form of an overlay:

[1]The **let** keyword binds a new variable. This construct can appear at any point within {...}-enclosed blocks. The new binding is in scope until the end of the block. One can add a type constraint using the : notation as shown in the example.

```
layout ipv6_header = {
  verpri         : overlay {
                     whole : 8
                   | parts : { version: 4,
                               priority: 4 }
                   },
  flow_label     : 24,
  ...
}
    ...
let x = pack[ipv6_header]
  [ verpri = [ whole = 0x60 ], ... ]
let y = pack[ipv6_header]
  [ verpri = [ parts = [ version = 6,
                         priority = 0 ] ], ... ]
```

Layouts can be composed on the fly in *layout expressions*. This can be useful when several small variations of the same basic layout are required. The following example assumes a 56-bit layout *lyt* that can appear at offsets 0, 16, or 24 within a 3-word (96 bit) tuple of packed data. Sequential layouts can be *concatenated* using the infix operator `##` and the notation $\{n\}$ specifies a small sequential layout that consists of nothing more than an unnamed *n*-bit *gap*:

```
layout lyt = { x: 16, y: 32, z: 8 }   // size = 56 bits
    ...
let udata =                    // pdata: word[3]
 if (...)                      // alignment is 0
    unpack[lyt ##{40}](pdata)
 else if (...)                 // alignment is 16
    unpack[{16}## lyt ##{24}](pdata)
 else                          // alignment is 24
    unpack[{24}## lyt ##{16}](pdata);
if (udata.x == 0x3456) ...
```

Notice that completely different instruction sequences must be generated for each of the three branches in order to correctly extract the values of bit fields like `udata.x`, but we are able to use the same layout definition *lyt* in each case.

3.3 Exposing Hardware Features

The IXP microengine instruction set is rather quirky, and Nova tries to shield the programmer from much of that. But some features are useful and we incorporated special syntax into Nova to let the programmer have access to them. Examples include inter-thread communication (on the same micro-engine, on different microengines on the same chip, or across chips), locking and unlocking (mutual exclusion), concurrency control, access to FIFOs, a number of special-purpose operations such as hashing, and access to different kinds of memory.

3.4 No GOTO

Nova provides syntax for the usual set of "structured" control constructs such as `if-then-else` or loops. However, it does not provide a general **goto** construct that would let the programmer specify arbitrary control flows. Nevertheless, most legitimate uses of **goto** have an alternative formulation in Nova in terms of function calls and exceptions. The Nova compiler translates all tail-calls into unconditional branches. In fact, the main difference between a tail call and a **goto** is that the tail-call respects scoping discipline and can pass arguments.

Programs running on an IXP microengine often implement no more than the *fast path* of a particular network protocol. As a result, the program must detect all cases that cannot be handled on the fast path. Once such a situation has occured, control should be transferred to some error handler immediately. The error handler could, for example, hand the current packet to the main CPU for slow-path processing.

To express such behavior in Nova, it is convenient to make use of the **try**-**handle** construct. Example:

```
fun g [ ..., x1, x2 ] {
  if (...) raise x2 ()
  else if (...) raise x1 [ b = ..., c = ... ];
  ...
}
...
try { if (x.a == A1) {  ...
        raise X1 [ b = ..., c = ... ]
      } else
        g [ ..., x2 = X2, x1 = X1 ]
} handle X1 [ b, c ] { ... }
  handle X2 () { ... }
```

Each **try**-**handle** block introduces—in a lexically scoped manner—the names of the exceptions (`X1` and `X2` in the example) that can be used within. These blocks can be nested, and exception values can be passed as arguments to function calls, thus enabling these functions (such as `g` in the example) to directly jump back out to the corresponding handler. Nova's type system guarantees that no computation that might raise an exception *e* can escape the **try**-**handle**-block that handles *e*.

4. THE FRONT END

Nova programs can never become very big because generated machine code has to fit within only a few thousand instruction words. Therefore, our compiler can easily afford to do whole-program analysis.

The front end of the compiler performs the usual lexical and syntactic analyses, elaboration and type-checking, and then converts to a *continuation-passing style* intermediate representation (CPS). It then further transforms the code into *static single assignment* (SSA) form for temporaries and performs a host of CPS optimizations, de-proceduralization, transformation to a new *static single use* form for temporaries participating in memory output operations, and, finally, IXP instruction selection. The phases after instruction selection are considered part of the back end.

4.1 Continuation-Passing Style

Continuation-passing style [23, 2] has been criticized as overkill in the case of a traditional stack-based implementation of languages with recursive procedures. Indeed, CPS makes it somewhat harder to determine which closures can be stack-allocated. On the other hand, some of the desirable properties of CPS (e.g., the explicit naming of all intermediate values) are shared with direct-style representations such as A-Normal Form [13]. Other studies point to heap-allocated continuation closures like those used by some existing CPS-based compilers such as SML/NJ [5] as being expensive [7].

However, none of these problems matter in the case of Nova: Nova does not have general recursion and its implementation is not stack-based. Due to the restrictions that we placed on the source language, closures do not require memory allocation and all free variables can simply remain in registers. Alternatives to CPS such as A-Normal Form do not capture control flow as concisely as does CPS. While we do not claim CPS to be strictly superior to direct-style representations, we still found it to be an excellent match for the problem of compiling the Nova language.

Our CPS does not have aggregate types; all variables conceptually correspond to single machine registers. The CPS converter takes advantage of information produced by the type checker and flattens all records, representing each leaf field by its own CPS variable. From that point on, each record field has its own independent representation and is subject to subsequent optimizations without regard of the conceptual relationship to other variables that had been expressed by types in the source program.

The converter also tries—as long as it is cheap to do so—to encode boolean values as control flow. In particular, functions returning a *bool* take two return continuations instead of one. This can make life somewhat easier for the CPS optimizer.

4.2 Static Single Assignment

Soon after CPS conversion the compiler eliminates all assignments to temporaries, effectively bringing the code into static single assignment form (for temporaries) [9]. Luckily, CPS is already powerful enough to express SSA directly without requiring additional constructs such as ϕ-nodes [18, 3].

As we will explain later (see Section 9), SSA is not only useful when it comes to performing data flow analyses. In our compiler its use guarantees an essential property: no variable will appear as the target of two different memory read instructions, ruling out the possibility of conflicting constraints that would make consistent colorings impossible. Although in principle it is possible to handle inconsistent colorings (i.e., the use of different colors at different program points), the resulting ILP models turned out to have too many variables and to cause solve times that are too long. The simplification of the ILP model resulting from the ability to rely on SSA is what makes our approach feasible.

4.3 De-proceduralization

The current prototype of our ILP-based back end cannot handle general interprocedural bank- and register-assignment. As a workaround we implemented a CPS phase that fully inlines all procedure calls in non-tail position.

4.4 CPS Optimization Phases

Our CPS optimizer is far from complete, but even now its output is good. We have implemented constant folding, global constant propagation, local value propagation, eta reduction, simple hoisting of arithmetic operations, simple contractions (e.g., inlining of called-once functions), useless variable elimination, dead code elimination, and trimming of memory reads.

In particular, the combination of flattened records, dead code elimination, and useless variable elimination makes programming with records and tuples (especially **pack** and **unpack**) inexpensive. Consider the following example:

```
fun f (p1, p2) {
  layout p = { a : 16, b : 32, c : 16 };
  let u1 = unpack[p](p1);
  let u2 = unpack[p](p2);
  (if (u1.c > 10) u1 else u2).b
}
```

Our compiler will determine that fields `u1.a`, `u2.a`, and `u2.c` are never used and that, therefore, their values do not even have to be extracted.

4.5 Static Single Use

As mentioned in section 4.2, SSA form solves a potential coloring problem for temporaries defined by memory read operations. A dual of the same problem arises for variables participating in memory write operations.

Therefore, just before going into instruction selection our compiler brings the program into *static single use* (SSU) form, a dual of SSA form where *cloning* plays the role of SSA's ϕ-nodes. For our purposes, static single use means that any use of a variable *x* as an operand in a memory-write operation is the *only* use of that variable in the entire program.

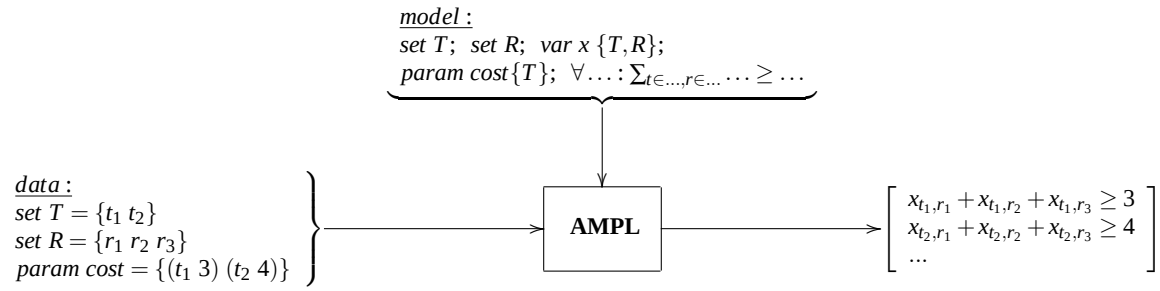


Figure 2: Modeling with AMPL. The input to AMPL consists of an abstract model and concrete data for that model. AMPL instantiates the model with the data, generates the (integer-)linear program to be solved by some off-the-shelf solver, and finally interprets the solution in terms of the original model.

SSU form can be generated simply by making sufficiently many copies of each variable. Since the program was in SSA form already, no variable will ever be written to after its creation, so originals and copies are guaranteed to be consistent.

We added a new primitive operation `clone` to the CPS language and to the IXP machine description used by the back end of the compiler. Cloning is semantically equivalent to copying. As we describe later in more detail, the difference is in how `clone` operations are handled by the ILP model: variables that are clones of each other *may* but *do not necessarily have to be* allocated to the same register, so cloning does not always imply physical copying. Minimizing the amount of such physical copying is part of the ILP solver's job, made possible by the idea that clones are copies that do not interfere with each other.

The example on the left is rewritten to that on the right where the variable `x` is cloned with `y` such that there is a single use of all members of the cloned set.

```
x ← ···              x ← ···
                     y ← clone(x)
   ...                  ...
sram() ← x           sram() ← y
   ...                  ...
   ← x                  ← x
```

5. ILP-BASED OPTIMIZATION

We model the problem of optimal bank-assignment and coloring of aggregates as a 0-1 integer-linear program, i.e., an optimization problem with constraints that are linear inequalities, a linear cost function, and the additional constraint that every variable must take the value 0 or 1. We use AMPL [14] to describe, generate, and solve the linear program. The AMPL compiler derives an instance of the optimization problem by instantiating a mathematical model with data specific to the task being solved, and feeds the resulting system to a standard off-the-shelf simplex solver.

The AMPL model consists of several variable-, set-, and parameter-declarations, plus templates to generate the constraints for the linear program. Sets can simply be symbolic enumerations, or they can be built up from other sets using constructive set operations. Related ILP variables are grouped together and referred to using a single name and a set that acts as an index range for that name. For example, if `T` and `R` are sets, then a declaration

```
var x {T,R};
```

introduces variables $x_{i,j}$ where `i` ranges over `T` and `j` over `R`. Figure 2 outlines an example of a model, data, and the generated system of linear equations.

5.1 Overview

In what follows, we refer to program variables in the code to be compiled as *temporaries* and reserve the word *variable* for the description of the AMPL model.

For every instruction of the program we want to find an assignment of temporaries to register banks. If a temporary is required in different banks for adjacent instructions i_1 and i_2, then an inter-bank move has to be inserted at the point between i_1 and i_2. In our model, there is a move for every live temporary at every point; if a temporary remains in a bank, then source and destination banks of the move are identical and its cost is zero.

A feasible solution is one that does not exceed any of the physical resource limits of the IXP such as number of registers in a bank. An *optimal* solution is one that has the lowest weighted cost of inter-bank moves.

5.2 The Model

We develop the model in three phases: basic constraints to express operand requirements and resource bounds, colors and aggregation of registers, and cloning. After we have covered the basics, each later progression will require additional or somewhat modified sets and constraints—which we will then introduce *on-the-fly*.

Variables: The set `V` is the set of temporaries in the program, `P` denotes the set of program points within the flowgraph. Each instruction of the program's original flowgraph is located *between* two such points. A branch instruction is followed by a single point that is connected to all points at the targets of the branch.

Using `P` and `V`, we then define sets related to liveness properties of the temporaries in the program. For any $v_1 \in$ `V` that is live at a point $p_1 \in$ `P`, we write $(p_1, v_1) \in$ `Exists`. The `Exists` set is similar to the live set but not identical: if an instruction between points p_1 and p_2 produces a result v that is immediately dead, then v is nowhere live but $(p_2, v) \in$ `Exists`. If a temporary v_1 is live and carried unchanged from point p_1 to p_2, then we say that $(p_1, p_2, v_1) \in$ `Copy`.

```
set Exists ⊆ P×V ;
set Copy ⊆ P×P×V ;
```

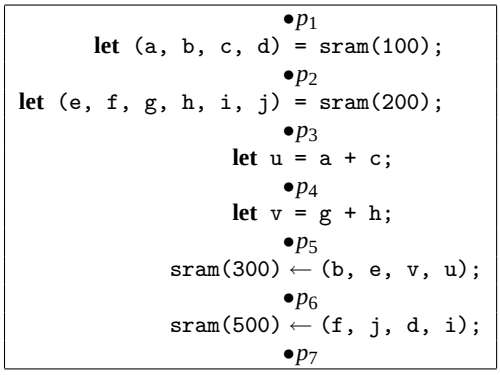
Our model starts with three sets of 0-1 variables:

$Move_{p,v,b1,b2}$ has the value `1` if the temporary `v` needs to be moved from bank `b1` to bank `b2` at the point `p`; and is zero otherwise.

$Before_{p,v,b}$ has the value `1` if the temporary `v` is in the bank `b` *before* the point `p`; and is zero otherwise.

$After_{p,v,b}$ has the value `1` if the temporary `v` is in the bank `b` *after* the point `p`; and is zero otherwise.

In AMPL these variables would be declared using:




```
set P  := {p1 p2 ... p7}
set V  := {a b c d e f g h i j u v}
set DefL4 := {(p1,p2,a,b,c,d)}
set DefL6 := {(p2,p3,e,f,g,h,i,j)}
set DefABW := {(p3,p4,u) (p4,p5,v)}
set Arith := {(p3,p4,a,c) (p4,p5,g,h)}
set UseS4 := {(p5,p6,b,e,v,u) (p6,p7,f,j,d,i)}
set Exist := {(p2,a), (p2, b), ...
              (p3,e), (p3, f), ...
              (p4, u), ...
set Copy := {(p2,p3,a) (p2,p3,b) ...
```

Figure 3: Sample source code and AMPL data. AMPL set definitions on the right capture the essence of the bank- and register-allocation problem for the program on the left.

```
set XBank := {L, LD, S, SD};
set GBank := {A, B, M};
set Banks := XBank ∪ GBank ;

var Move {Exists, Banks, Banks} binary;
var Before {Exists, Banks} binary;
var After {Exists, Banks} binary;
```

where the set declarations enumerate the transfer banks in `XBank` and general purpose banks in `GBank`. The declaration for *Move* defines a variable indexed over points p and temporaries v such that $(p,v) \in \texttt{Exists}$, and a pair of banks representing the source and destination banks.

Instruction operands: Like Appel and George [4], we characterize instructions by the resources they require and define. For example, the output of the ALU can either be an **A**/**B** register or one of the write-side transfer registers. The operands can be taken from any disjoint set of input banks. Thus an instruction of the form $x := y \oplus z$ between the points p_1 and p_2 will be modeled as $(p_1,p_2,x) \in \texttt{DefABW}$, and $(p_1,p_2,y,z) \in \texttt{Arith}$, and constraints on these sets will ensure that the necessary conditions for the instruction are met.

```
set DefABW ⊆ P × P × V
set Arith  ⊆ P × P × V × V
```

In a similar manner we have a number of sets to characterize the various operand- and destination constraints of other IXP instructions.

On the IXP, multiple memory locations can be read or written using a single instruction, and all the registers used in these operations must be consecutive. The number of registers used in SDRAM memory operations is always a multiple of two. We declare the sets $\texttt{DefL}_i, \texttt{UseS}_i : 1 \le i \le 8$, and $\texttt{DefLD}_j, \texttt{UseSD}_j : j \in 2,4,6,8$ to associate the points before and after the instruction with the variables defined or consumed by it. For example:

$$\texttt{set DefL}_i \ \subseteq \texttt{P} \times \texttt{P} \times \overbrace{\texttt{V} \times \cdots \times \texttt{V}}^{i}$$

Others: Our model uses several other sets that, for brevity, are not shown in this extended abstract. The purpose of these sets is to deal with, e.g., instructions that mutate an operand or situations where it would be illegal to insert move instructions at certain program points.

Example: Figure 3 shows a sample program and the AMPL data generated for it. The first two lines read four and six SRAM memory locations from addresses 100 and 200; the last two lines, write values to addresses 300 and 500 in SRAM. The AMP data declares 7 programs points and 12 variables; `(`p_3`,`p_4`,u)` is a member of `defABW` since there is an arithmetic operation between p_3 and p_4, and the destination of the instruction is `u`, etc.

6. CONSTRAINTS

Linear constraints deal with liveness, operand constraints, and **K** constraints which guarantee that at no program point the number of temporaries assigned to a bank exceeds the capacity of that bank. Finally, there are constraints that bind together the *redundant* variables of the model. (Redundant variables are variables that exist merely to make it easier to specify the model but whose values are uniquely determined by the values of other variables.)

In-before and in-after: If at some point p a temporary v is moved from some source bank b_1 to some other bank b_2 (even when $b_1 = b_2$), then v must have existed in b_1 before p and it must exist in b_2 after p. The constraints relate *Before* and *After* to *Move*s:

$$\forall (p,v) \in \texttt{Exists},\ \forall b \in \texttt{Banks}: \\ \mathit{Before}_{p,v,b} = \textstyle\sum_{d \in \texttt{Banks}} \mathit{Move}_{p,v,b,d}$$

$$\forall (p,v) \in \texttt{Exists},\ \forall b \in \texttt{Banks}: \\ \mathit{After}_{p,v,b} = \textstyle\sum_{s \in \texttt{Banks}} \mathit{Move}_{p,v,s,b}$$

In one place only: We place the restriction that if a temporary exists at a point it must exist in precisely one bank. Therefore, the sum over all banks must be one. This constraint must be relaxed when cloning is involved, however for *non-cloned* variables, the in-one-place assumption simplifies the modelling, and does not appear to impact optimality in practise:

$$\forall (p,v) \in \texttt{Exists}: \sum_{b \in \texttt{Banks}} \mathit{Before}_{p,v,b} = 1$$

Copy propagation: If a temporary is copied unchanged between p_1 and p_2, then its location after p_1 must be the same as its location before p_2: This constraint propagates liveness information and is expressed as:

$$\forall (p_1,p_2,v) \in \texttt{Copy},\ \forall b \in \texttt{Banks}: \mathit{After}_{p_1,v,b} = \mathit{Before}_{p_2,v,b}$$

Operand definition: Sets like `DefABW` describe results of instructions with more than one possible destination bank. Here, the destination may be in the **A**, **B**, **S**, or **SD** bank just before the point following the instruction. Since an operand can only be in one bank at a time, the corresponding summations must equal one.

$$\forall (p_1,p_2,v) \in \texttt{DefABW}: \\ \mathit{Before}_{p_2,v,\mathbf{A}} + \mathit{Before}_{p_2,v,\mathbf{B}} + \\ \mathit{Before}_{p_2,v,\mathbf{S}} + \mathit{Before}_{p_2,v,\mathbf{SD}} = 1$$

Arithmetic: If x and y are involved in an ALU operation, then they must come from one of the input register banks **A**, **B**, **L**, or **LD**:

$$\forall(p_1,p_2,x,y) \in \texttt{Arith} : \sum_{b\in\{\mathbf{A},\mathbf{B},\mathbf{L},\mathbf{LD}\}} After_{p_1,x,b} = 1$$
$$\forall(p_1,p_2,x,y) \in \texttt{Arith} : \sum_{b\in\{\mathbf{A},\mathbf{B},\mathbf{L},\mathbf{LD}\}} After_{p_1,y,b} = 1$$

However, x and y cannot be in the same register bank:

$$\forall(p_1,p_2,x,y) \in \texttt{Arith}, b \in \{\mathbf{A},\mathbf{B},\mathbf{L},\mathbf{LD}\} : \\ After_{p_1,x,b} + After_{p_1,y,b} \leq 1$$

Furthermore, if one of the operands is in a transfer bank, then the other operand cannot be in a transfer bank:

$$\forall(p_1,p_2,x,y) \in \texttt{Arith} : After_{p_1,x,\mathbf{L}} + After_{p_1,y,\mathbf{LD}} \leq 1$$
$$\forall(p_1,p_2,x,y) \in \texttt{Arith} : After_{p_1,x,\mathbf{LD}} + After_{p_1,y,\mathbf{L}} \leq 1$$

Aggregate definition and use: The aggregate definition and use constraints such as $\texttt{DefL}_i$ and $\texttt{UseS}_i$ are similar; the constraints merely cycle through the variables involved in the aggregate, specifying where they should exist. We only show the case $i = 4$:

$$\forall(p_1,p_2,v_1,v_2,v_3,v_4) \in \texttt{DefL4}, \\ \forall i \in 1..4 : \quad Before_{p_2,v_i,\mathbf{L}} = 1$$

$$\forall(p_1,p_2,v_1,v_2,v_3,v_4) \in \texttt{UseS4}, \\ \forall i \in 1..4 : \quad After_{p_1,v_i,\mathbf{L}} = 1$$

K and Spilling for A/B: The ILP model does not pick colors for temporaries in **A**/**B** banks; this is left to a subsequent coloring phase. To prevent that phase from having to insert additional inter-bank moves or spills, the model makes sure that no more than 16 **A**/**B** registers are needed at any time. We leave room for one extra register in **A** to be able to implement cycles in parallel copies that might be needed during optimistic coalescing [4]. The **K** constraints are needed both before and after each point; for brevity we only show the former kind:

$$\forall p \in P : \sum_{(p,v)\in\texttt{Exists}} Before_{p,v,\mathbf{A}} \leq 15$$

$$\forall p \in P : \sum_{(p,v)\in\texttt{Exists}} Before_{p,v,\mathbf{B}} \leq 16$$

The **K** constraints for transfer banks is deferred to Section 9.

7. OBJECTIVE FUNCTION

The objective is to minimize the weighted cost of moves. For each point we compute a static frequency estimation based on loop nesting and branch probabilities using the Dempster-Shaffer theory to combine probabilities. (Our own variation of the Wu-Larus frequency estimation [25] can cope with irreducible flowgraphs.) The objective function uses the following parameters:

```
param weight{P};  /* execution freq */
param mvC := 1;         /* move cost */
param ldC := 200;       /* load cost */
param stC := 200;       /* store cost */
param bias := 1.01;
```

mvC is the cost of a register-register move, and stC and ldC are the cost of accessing spill memory.

If $Move_{p,v,\mathbf{A},\mathbf{M}} = 1$, then the move from **A** to **M** will be implemented as a register-register move from $\mathbf{A} \rightarrow \mathbf{S}$, followed by a $(\mathbf{S} \rightarrow \mathbf{M})$. This is reflected in the objective function by using the term $(mvC + stC) \cdot Move_{p,v,\mathbf{A},\mathbf{M}}$. (We also added a small bias towards using **A** registers over **B** registers since we found that this speeds up the ILP solver.) Here is the portion of the objective function that is related to moves from **A** and **B**.

$$\begin{aligned}
&\sum_{(p,v)\in\texttt{Exists}}(\\
&\quad /* \text{ from } \mathbf{A} \text{ bank } */ \\
&\quad \texttt{weight}_p \cdot \texttt{mvC} \cdot \sum_{b\in\{\mathbf{B},\mathbf{S},\mathbf{SD}\}} Move_{p,v,\mathbf{A},b} \\
&+ \quad \texttt{weight}_p \cdot (\texttt{mvC} + \texttt{stC}) \cdot Move_{p,v,\mathbf{A},\mathbf{M}} \\
&+ \quad \texttt{weight}_p \cdot (\texttt{mvC} + \texttt{stC} + \texttt{ldC}) \cdot Move_{p,v,\mathbf{A},\mathbf{L}} \\
&\quad /* \text{ from } \mathbf{B} \text{ bank } */ \\
&+ \quad \texttt{bias} \cdot \texttt{weight}_p \cdot \texttt{mvC} \cdot \sum_{b\in\{\mathbf{A},\mathbf{S},\mathbf{SD}\}} \cdot Move_{p,v,\mathbf{B},b} \\
&+ \quad \cdots)
\end{aligned}$$

8. A MILLION VARIABLES

The above description of the model was a simplification. In practice, if defined this way, problem sizes grow too large and cannot be solved with reasonable resources. Since there are 7 banks, the number of *Move* variables for each temporary at each point is 7^2. Since there are 64 available registers, we could theoretically have that many live temporaries at every program point. Even if we assumed just an average of 20 live variables at every point and a full instruction cache of 1000 instructions, then we would have approximately one million *Move* variables ($7^2 \cdot 20 \cdot 1000$), not to mention *Before*, *After*, and so on.

To reduce the number of variables we perform a static analysis on the use of temporaries. For example, if a temporary is loaded from SRAM memory and is never stored back anywhere, then there is no reason for it to ever be in **S**, **SD**, or **LD**. We will therefore rule out all transitions to and from these banks. Of course, if the temporary is spilled, then it will have to make a brief appearance in **S**. Ruling out these banks implies that spilling will move the temporary either from {**L**, **A**, **B**} directly to **M**, and reloading will move from **M** directly to {**L**, **A**, **B**}. Since spilling occurs very rarely, these restrictions, which result in dramatically smaller optimization problems, are not a problem in practise.

9. AGGREGATES AND COLORING

In the work of Appel and George the program generated from the results of integer-linear programming satisfied the **K** constraints, and subsequent coloring phases were used to assign registers using a variation of the Park and Moon [20] optimistic coalescing. We use the same approach for the **A** and **B** bank, but for transfer registers this is not possible because of aggregation.

In the example shown in Figure 3, two temporaries from the first read must be moved out of the transfer bank to make room for the next read. If a naïve "optimal" solution to the coloring problem does this in a manner resulting in fragmentation of the bank, then the second read cannot be performed even though a **K**-constraint was satisfied.

Our approach is to let the ILP solver derive a coloring of aggregates directly in conjunction with the bank assignment problem. For this we added the following declarations to the model:

```
set XRegs := 0..7;
var Color {V, XBank, XRegs} binary;
```

XRegs is an enumeration of register numbers, $Color_{v,b,r} = 1$ if whenever v is in the transfer bank b, then it is in the register r of that transfer bank. There are a couple of important properties of the *Color* variable that are in the spirit of Fu and Wilken[15], in that the number of variables and constraints are manageable, (but optimality, in our case, may be compromized):

1. Color is point-independent: whenever a variable is in a specific bank, it will always reside in the same registers (even

after spilling/reloading).

2. The objective function is color-independent.
3. In general, if the flowgraph were not in static single assignment, the problem could become unsolvable. Consider a program:

```
(a,b,X,Y) ← sram(...)
(Y,X,u,v) ← sram(...)
```

   In the first read operation `Y` must be in a higher numbered register than `X`, the opposite is true for the second operation, so there is no feasible solution. But programs like this are not in SSA form, and cannot occur.
4. The analogous problem on the write-side is avoided by the use of static single use form (Section 10). Assuming a bank of size four, without static single use form, there would be no solution for:

```
sram(...) ← (X, a, b, c);
sram(...) ← (a, b, c, X);
```

For continuity we express the constraints below using $\forall(p,v) \in$ *Exists* and do not burden the reader with narrowing this down as described in Section 8.

Color: There are three constraints related to colors. A color must exist for a temporary that can live in a transfer bank:

$$\forall v \in \mathtt{V}\ \ b \in \mathtt{XBank}: \ \sum_{r \in \mathtt{XRegs}} Color_{v,b,r} = 1$$

If two interfering temporaries are simultaneously live in a transfer bank, then they must not have the same color. Our model lists interferences explicitly in a set $\mathtt{Interferes} \subseteq \mathtt{V} \times \mathtt{V}$ because (due to cloning—see Section 10) it is not always true that two temporaries interfere just because they are both live at the same time:

$$\forall (p,v_1),(p,v_2) \in \mathtt{Exists}, (v_1,v_2) \in \mathtt{Interferes}$$
$$\forall b \in \mathtt{XBank}\ \ \forall r \in \mathtt{XRegs}:$$
$$Before_{p,v_1,b} + Before_{p,v_2,b} + Color_{v_1,b,r} + Color_{v_2,b,r} \leq 3$$

An analogous constraint covers the *After* case.

Aggregation: The bulk of the constraints in our system have to do with aggregation, and they are all fairly simple and similar in nature. Adjacency is expressed pairwise:

$$\forall (p_1,p_2,\ldots,v_k,v_{k+1},\ldots) \in \mathtt{DefL}_i\ \forall r \leq 7-i+k:$$
$$Color_{v_k,\mathbf{L},r} = Color_{v_{k+1},\mathbf{L},r+1}$$

We found that adding a redundant set of constraints that immediately rules out a number of impossible allocations for an aggregate speeds up the solver. For example, the first temporary in an aggregate of three cannot possibly have colors 6 or 7.

Same register: Some instructions use one register number to refer to two different registers in two different banks. When coloring the corresponding temporaries we must make sure that they end up having the same color. Consider:

```
dst ← hash(src)
dst ← (sram[addr=ea,bit_test_set] ← src)
```

In the first line `dst` gets the hash of `src`, and in the second the memory at effective address `ea` is modified by `src`, and the old value returned in `dst`. Each of these operations correspond to instructions on the IXP, and in each case `dst` and `src` have the same register number but are in different transfer banks. We use a set

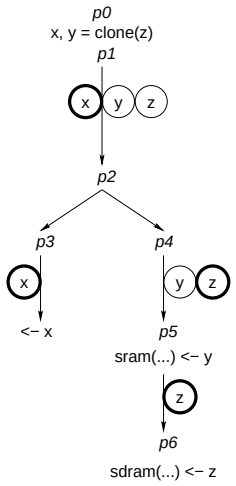


Figure 4: An example of cloning. *x* has clones *y* and *z*. All three start out at the same location but eventually have to split up to satisfy other constraints.

`SameReg` whose members are pairs of temporaries together with the two points just before and after the instruction in question.

$$\mathtt{set\ SameReg} \subseteq \mathtt{P} \times \mathtt{P} \times \mathtt{V} \times \mathtt{V}$$

$$\forall (p_1,p_2,d,s) \in \mathtt{SameReg}\ \ \forall r \in \mathtt{XRegs}:$$
$$Color_{d,\mathbf{L},r} = Color_{s,\mathbf{S},r}$$

K and Spilling for transfer banks: We do not need **K** constraints for **LD** or **SD** because the fact that a color must be picked for each occupant of the bank effectively limits their number. Even though the model also colors **L** and **S**, the situation here is slightly different because we sometimes need an extra register for implementing spills. To handle this, we added a set of 0-1 variables $colorAvail_{p,b,r}$ for $b \in \{\mathbf{L},\mathbf{S}\}$ which indicate whether or not at point p one could allocate some variable to register r in b:

$$\forall v: Color_{v,b,r} + Before_{p,v,b} \leq 1 + colorAvail_{p,b,r}$$

The **K** constraint for **L** and **S** then becomes

$$\sum_{r \in \mathtt{XRegs}} colorAvail_{p,b,r} = 8 - needsSpill_{p,b}$$

where $needsSpill_{p,b}$ indicates that spilling requires a spare register in bank b at point p. $needsSpill_{p,b}$ itself is constrained by inequalities of the form

$$needsSpill_{p,b} \geq Move_{p,v,b_1,b_2}$$
$$needsSpill_{p,b} \leq \sum_{b_1,b_2:\ldots} Move_{p,v,b_1,b_2}$$

for all relevant combinations of b_1 and b_2. We found that the second constraint (which is not necessary for correctness) improves solve times by tightening the model somewhat:

10. THE ROLE OF STATIC SINGLE USE

In Figure 4, *x* has clones *y* and *z*. All of them eventually get used in different contexts. Immediately after a `clone` instruction, the original and all its clones are still in the same register. For our model this means that they are in the same bank *b* and, if *b* is a transfer bank, their *b*-colors are the same. The members

(p_1, p_2, d, s) of the `Clone` set indicate that a clone instruction exists between p_1 and p_2, cloning the source s resulting in d. With this we can write:

$$\forall (p_1, p_2, d, s) \in \texttt{Clone} : \mathit{Before}_{p_2,d,b} = \mathit{After}_{p_1,s,b}$$
$$\forall (p_1, p_2, d, s) \in \texttt{Clone}, r_1 \neq r_2, b \in \texttt{XBank} :$$
$$\mathit{Color}_{d,b,r_1} + \mathit{Color}_{s,b,r_2} \leq 2 - \mathit{Before}_{p_2,d,b}$$

Other than at the point of cloning itself, there is no requirement for cloned temporaries to exist in the same bank or the same register. In our example (Figure 4) all three variables eventually must—and will—move to different banks to satisfy other constraints. In particular, z could get moved to its final destination even before p_5 so that y and z exist in different banks simultaneously at that point (y in **S** and z in **SD**). But y and z represent the same program variable, so the cloning device effectively allows for such a program variable to be in more than one place at the same time.

We need to make the following adjustments in our model to account for the possibility of cloning:

- When coloring, we require different colors for live temporaries that *interfere* with one another. By definition, temporaries that are each other's clones do not interfere.
- In **K** constraints for **A**/**B** we should count only one representative for each set of mutual clones that is in the respective bank. For example, if x, y and z are all in **A** at p_2, then they will all be in the same register.
- In the objective function we should count as one any collection of moves that involve members of the same clone set, moving them from identical sources to identical destinations.

We take care of the first point by guaranteeing that whenever x and y are clones of one another, then neither $(x,y) \in \texttt{Interferes}$ nor $(y,x) \in \texttt{Interferes}$.

The other two points require a different way of counting. For this we define three new sets of 0-1 variables referred to as *cloneBefore*, *cloneAfter*, and *cloneMove* and include constraints that tie them to *Before*, *After*, and *Move* as follows:

First we look at each program point p and consider the sets $\{x_1, \ldots, x_n\}$ of temporaries that are clones of each other and live at p. From $\{x_1, \ldots, x_n\}$ we pick a *representative* x_r for the point p and require that $\mathit{cloneBefore}_{p,x_r,b} = 1$ if and only if there is at least one $x \in \{x_1, \ldots, x_n\}$ so that $\mathit{Before}_{p,x,b} = 1$. This can be expressed roughly as follows:

$$\forall x \in \{x_1, \ldots, x_n\} : \mathit{cloneBefore}_{p,x_r,b} \;\geq\; \mathit{Before}_{p,x,b}$$
$$\mathit{cloneBefore}_{p,x_r,b} \;\leq\; \textstyle\sum_{x \in \{x_1,\ldots,x_n\}} \mathit{Before}_{p,x,b}$$

The same definition, *mutatis mutandis*, works for *cloneAfter* and also for *cloneMove*.

With this, **K** constraints can be adjusted to look at *cloneBefore* and *cloneAfter* instead of *Before* or *After* when dealing with clones. A similar change is done in the objective function using *cloneMove*.

In the example (see Figure 4) we have x to be the representative for x, y, and z between p_1 and p_2. Later on, between p_4 and p_5, x is not live any more, so a different representative gets used there.

11. RESULTS

Network processors are so new that a set of standard benchmarks does not exist. We have exercised our compiler on a large number of problems, however three programs stand out both in size and in real-world relevance.

AES Rijndael: implements the NIST standard for encryption based on Rijndael [10, 11]. Our implementation has the following variation from the fast C reference implementation available from `http://www.nist.gov`.

- We keep the encryption state in registers at all times, sometimes exploding 4 registers holding the state into 16 registers containing the individual bytes.
- The ethernet, IP, and TCP headers are shifted before encryption so that plaintext is read potentially quad-word misaligned, but the ciphertext is written out quad-word aligned.
- The code maintains the TCP checksum field.
- All tables reside in SRAM memory, resulting in contention; we did not investigate distributing the tables over the different memory banks.
- The key expansion was statically computed.
- We did not implement CBC, so the data size must be a multiple of 16 bytes.

Kasumi: implements the Kasumi encryption algorithm[1] used in the ETSI 3GPP standard. Like Rijndael, this implementation shifts headers, statically computes the subkey expansion, and maintains the TCP checksum. All tables are stored in scratch memory, except the S9 table, which is stored in SRAM memory. By interleaving and packing all the subkey tables, each iteration performs one scratch read to access all the 16 subkey elements.

IPv6-IPv4 NAT This program implements network address translation (NAT) between IPv6 and IPv4 headers [17]. Because of the different header sizes, the start of the packet must be moved to a new location and care is required in updating the new checksum field.

In all cases the code for the application must be compiled with code that synchronizes with the receive scheduler, reads in the packet from the receive FIFOs to SDRAM memory, synchronizes with the transmit scheduler, and contains the logic to invoke the worker thread.

Figure 5 summarizes the characteristics of these programs. Line counts are those reported by `wc` and includes whitespaces and comments. The numbers for NAT are those for an older (and now obsolete) version of Nova that was lower-level and did not have layouts.

Figure 6 shows the part of the AMPL statistics related to the number of variables that participate in coloring. All the variables mentioned in `DefLi` were added up and included in the first column. The table shows that the model has to deal with a fair deal of coloring.

Next we show the time it takes to solve the root relaxation (optimal linear solution), the time it takes to find the optimal integer solution (within 0.01% of optimal), the size of the optimization problem, and the number of inter-bank moves and spills. The numbers are in Figure 7; there are too few data points at the present time to plot a trend, but our experimentation gives reason to be optimistic about solve times. All numbers are for CPLEX [8] on an 800MHz dual pentium-III processor, 2Gbyte, Linux machine.

We have experimented with another objective function that lets us determine whether spills are required at all, and if so, where. Once this has been determined many of the variables and constraints involving memory can be eliminated, resulting in a much smaller linear program. We have not found it necessary to follow this route (which gave solve times of 9 seconds for AES and 19.2

	Line count		Layouts			Exceptions	
	Nova	**instructions**	**specs**	**pack**	**unpack**	**raise**	**handle**
AES	541	588	7	8	5	3	1
Kasumi	587	538	7	7	4	2	2
NAT	839	740	-	-	-	-	-

Figure 5: Static benchmark program statistics

	`DefLi`	`DefLDj`	Total	`UseSi`	`UseSDj`	Total
AES	68	16	84	4	10	14
Kasumi	44	14	58	4	14	18
NAT	43	22	65	8	60	64

Figure 6: AMPL statistics

	Solve Time (sec)		Variables	Constraints	Terms in Objective	Solution	
	Root	**Integer**	×1000	×1000	×1000	**Moves**	**Spills**
AES	30.4	35.9	108	102	37	25	0
Kasumi	48.2	59.2	138	131	50	20	0
NAT	69.2	155.6	208	203	72	60	0

Figure 7: Solver statistics

seconds for NAT), and a detailed description is beyond the scope of this paper.

Lastly we have exercised the output of our compiler on real in-house hardware [22] consisting of a 233 MHz IXP1200 with data from a hardware packet generator. For Rijndael we measured 270Mbs for payloads of 16 bytes, and 320, 210, and 60 Mbs for 8, 16, and 256 byte payloads using Kasumi. None of these programs were written to be highly optimized for bit-rate processing speeds.

12. FUTURE WORK / OPEN PROBLEMS

There are many opportunities to improve the current design and implementation and more open research questions to address:

newer IXP chips Our project focused on the IXP1200. While most of the ideas will carry over to more recent versions of the IXP hardware, there are also a number of new features (e.g., nearest neighbor registers, and signals) that need to be addressed.

multithreading Nova provides a key piece of the puzzle, however a complete solution must address the issues of *co-operative* multithreading among micro-engines, threads, and host processor.

module system The addition of a module system together with a more powerful type system (e.g., one that lets the programmer create new abstract types) will become more important as programmers begin to collect larger utility libraries of IXP code.

re-materialization A compiler with aggressive constant folding and propagation such as ours tends to create intermediate code with many residual constants. Loading a constant is not without cost: on the IXP it takes 1 or 2 instructions, depending on the value. To avoid wasting cycles this way one can keep frequently used constants in registers. But this can be costly, too, because it will increase register pressure. Thus, the problem of loading constants should be solved by the register allocator, and a simple trick makes this straightforward. We treat every individual constant as a temporary and invent a virtual register bank **C**. **C** has unlimited capacity and can hold constants (but nothing else). A move to **C** represents the operation of discarding a constant from a physical register; it has zero cost. A move from **C** represents the load operation of the corresponding constant; its cost depends on the value of the constant (which is statically known). This scheme can be further refined by paying attention to pairs (c_1, c_2) of constants where calculating c_2 from c_1 is cheaper than loading c_2 from scratch. (We have an AMPL model that takes all this into account, but we did not find the time to complete the rest of our compiler infrastructure to take advantage of it.)

global register allocation Our compiler fully inlines all procedure calls, and Nova was designed for this to always be possible. However, excessive inlining can easily cause code explosion. A better compilation scheme would be to compile to a run-time model that permits genuine procedure calls. But on the IXP1200 we cannot afford to use a stack. The challenge is then to find an ILP formulation of register allocation that derives globally optimal calling conventions and callee-save assignments for every procedure [6].

13. CONCLUSIONS

Unconventional architectures with compilation problems that do not have good heuristic solutions require unconventional compilation techniques. Our results indicate that 0-1 integer linear programs can provide an excellent solution for an architecture such as the Intel IXP. Our ILP formulation addresses three open problems without good known heuristics: bank assignment, coloring of aggregates in conjunction with bank assignment, and the management of variables in multiple locations.

The formulation of our model turns out to be straightforward. Although not optimal in the strict sense, we have evidence to believe that the solutions are very good. Finding a correct mathematical model is relatively easy, but we found that engineering such a model to reduce the number of redundant variables and constraints is extremely important. Making the right set of assumptions to limit

the number of variables and constraints is critical—as illustrated by the color- and clone-related variables.

More experimental results are required to evaluate the performance of the model for problems generated by the compiler. But since the model is similar in flavor to that used by Appel and George (where they show that solve times for over 600 flowgraphs, some having thousands of instructions, were always within 30 sec), we can be cautiously optimistic. Our situation is more complex because we deal with a considerably more difficult problem. On the positive side, though, we know that IXP programs cannot grow much bigger than the ones we tried successfully.

Our programming language provides the tools for writing robust and maintainable programs, especially when comparing it with the current state of the art: assembler or Intel's C compiler for the IXP. Tuples, layouts, loops, functions and exceptions appear to provide the right balance of expressiveness and efficiency. CPS turned out to be a great intermediate representation for the kind of language that does not require memory-allocated closures.

14. ACKNOWLEDGEMENTS

We thank *Satish Chandra* for one of the earlier Nova front ends; *John Reppy* for the control flow graph frequency estimation and as one of the initiators of the Nova project; *David Gay* for help with AMPL and solvers; *Ron Sharp* and *Mike Coss* for comments and help with the IXP architecture, simulator, and Tadpole board[22]; and *Andrew Appel*, *Cliff Young*, and the anonymous referees for comments on an earlier draft.

15. REFERENCES

[1] 3GPP. Specification of the 3GPP confidentiality and integrity algorithms. Version 1.2, Sept. 2000.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[3] A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, April 1998.

[4] A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–253, 2001.

[5] A. W. Appel and D. B. MacQueen. A Standard ML compiler. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 301–24, New York, 1987. Springer-Verlag.

[6] U. Boquist. Interprocedural register allocation for lazy functional languages. In *Proceedings of the 1995 Conference on Functional Programming Languages and Computer Architecture (FPCA)*, La Jolla, California, USA, June 1995.

[7] W. D. Clinger, A. Hartheimer, and E. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

[8] CPLEX mixed integer solver. www.cplex.com, 2000.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, October 1991.

[10] J. Daemen and V. Rijmen. *The block cipher rijndael*, pages 288–296. LNCS 1820. Springer-Verlag, 2000. J.-J.Quisquater and B.Schneier, Eds.

[11] J. Daemen and V. Rijmen. Rijndael, the advanced encryption standard. *Dr. Dobb's journal*, 26(3):137–139, March 2001.

[12] J. R. Ellis. *Bulldog: A compiler for VLIW architectures*. The MIT Press, 1986.

[13] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, New York, 1993. ACM Press.

[14] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, South San Francisco, CA, 1993.

[15] C. Fu and K. Wilken. A faster optimal register allocator. In *The 35th Annual IEEE/ACM International Symp. on Microarchitecture*. IEEE/ACM, November 2002.

[16] L. George and A. W. Appel. Iterated register coalescing. *ACM transacations on programming languages and systems.*, 18(3):300–324, May 1996.

[17] E. Grosse and Lakshman Y. N. Network processors applied to IPv4/IPv6 transition. Bell Labs Report.

[18] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations*, volume 30, pages 13–22. ACM Press, Mar. 1995.

[19] T. Kong and K. D. Wilken. Precise register allocation for irregular architectures. In *31st International Microarchitecture Conference*. ACM, December 1998.

[20] J. Park and S.-M. Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architecture and Compilation Techniques*, pages 196–204, 1998.

[21] B. Scholz and E. Eckstein. Register allocation for irregular architectures. In *LCTES/SCOPES*. ACM, June 2002.

[22] R. Sharp, M. Blott, M. Coss, B. Ellis, D. Majette, and V. Purohit. Starburst: Building next-generation internet devices. *Bell Labs Technical Journal*, 6(2):6–17, 2001.

[23] C. Strachey and C. Wadsworth. Continuations: A mathematical semantics which can deal with full jumps. Technical Monograph PRG-11, Programming Research Group, Oxford University, 1974.

[24] S. Tallam and R. Gupta. Bitwidth aware global register allocation. In *30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–96. ACM, January 2003.

[25] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *27th IEEE/ACM Inter'l Symp. on microarchitecture (MICRO-27)*. IEEE/ACM, Nov. 1994.