ALTO

A Link-Time Optimizer for the DEC Alpha
$Revision: 1.1 $
$Date: 1998/02/13 19:41:28 $
$Generated: 7 April 1998$

Robert Muth

1 Common Abbreviations

bbl	basic block
edg	edge
fdr	file descriptor
frm	frame
fun	function
ins	instruction
reg	register
regs	register set
pdr	procedure descriptor
rel	relocation
sec	section
stk	stack
sym	symbol

2 Manpages for ALTO and Related Tools

2.1 Alto Manpage

Invocation

alto *option**

Options

The following *option*s are available. Please note, that each option should appear at most once. Otherwise, the latest occurence will overwrite all previous ones.

-p
: Load pixie profiling information. The filnames are assumed to be the the input file with ".Addrs" or ".Counts" appended. Currently, we only support the new pixie format generated by the latest pixie version.

  The latest version of OSF/1 (Digital UNIX V4.0) uses atom to emulate pixie. Alto is tuned for the use with pixie profiling information so use it whenever possible

-r [*number-of-rounds*] **(default 5)**
: Specifies how many rounds the optimizer should run the "easy optimization" at most. The c-compiler shipped with the latest version of OSF/1 (Digital UNIX V4.0) creates binaries that alto cannot handle unless rounds > 0. By "easy optimizations" we mean non-inlining optimization. Currently, alto does "easy optimization" first, then inlining, and finally "easy optimizations" again.

-i [*input-file*] **(default a.out)**
: Specifies the name of the file that should be optimized.

-o [*output-file*] **(default b.out)**
: Specifies the name of the file being generated.

-O [*optimize-options*] **(default none disabled)**
: Disables certain optimizations see Chapter 8 [OPT], page 31

  If you want to disbable more that one optimization concatenate the corresponding options with an arbitary delimiter, eg. '+'.

  The complete list of optimizations is changing rapidly. An uptodate overview can be obtained by running the following command in the master source directory.

  '`grep DISABLED_OPT *.c`'

-P [*print-options*] **(default none enabled)**
: Enables the output of certain information.

  If you want to print out more that one item concatenate the corresponding options with an arbitary delimiter, eg. '+'.

The complete list of print item is changing rapidly. An uptodate overview can be obtained by running the following command in the master source directory.

'grep PRINT_OPT_WHAT *.c'

-T [*time-options*] **(default "postopt")**

This flag lets you choose when to print the print items selected with the -P option. Default is after all optimizations are done.

If you want to print out items at more than one point concatenate the corresponding options with an arbitary delimiter, eg. '+'.

The complete list of time options is changing rapidly. An uptodate overview can be obtained by running the following command in the master source directory.

'grep PRINT_OPT_WHEN *.c'

-M [*inline-method*] **(default Crit)**

This flag selects the inline method used by alto. Possible values are: Crit (default),Path,Small, And No.

The complete list of methods is changing. An uptodate overview can be obtained by running the following command in the master source directory.

'grep INLINE_METHOD *.c'

Preparing Binaries For Processing with ALTO

In order to optimize binaries with alto they have to be linked using the flags listed below. Unfortunately, this does not quite work for the gnu compiler due to a stupid bug that might have been fixed by now. the problem is that gcc and g++ pass the -r flag (from -W,-r) too late to the linker.

Here is a workaround. Run "gcc -v" this tells you which spec file gcc is using. Patch the spec file as follows. Goto the line following "*link:". make "-r " the first three characters in this line. Omit the option "-Wl,-r" in the table below.

compiler	**option**
cc	-Wl,-r -Wl,-d -Wl,-z -non_shared
f77	-Wl,-r -Wl,-d -Wl,-z -non_shared
gcc	-Wl,-r -Wl,-d -Wl,-z -static
g++	-Wl,-r -Wl,-d -Wl,-z -static

Example Invocations

alto Runs alto on binary a.out producing an (optimized) binary b.out. No profiling information is used.

alto -i a.rr -o my.exe

Runs alto on binary a.rr producing an (optimized) binary my.exe.

alto -p -i a.rr -o my.exe
: As before but profiling information is used and expected to be provided in the files a.rr.Addrs and a.rr.Counts

alto -O Load+Branchtwist
: Runs alto as usual on a.out but does not perform load-avoiding and conditional-branch-twist optimizations.

(Known) Bugs

Alto assumes that the GP register has the same value throughout the program.

Alto assumes that all branches have an offset of less than 2^20=1M instructions.

2.2 Sloop Manpage

Introduction

Sloop reads in a description of a flow graph for the functions in a program (usually generated by alto using the "-P loop" option) and prints out the graphs and/or subgraphs for selected functions, loops, or basic blocks, as specified using the options below. The critical path(s) in the (sub)graph are displayed in green, while any function call out of a basic block is indicated in red.

Invocation

sloop [option*]

Options

-I Print instructions within basic blocks. Normally, instructions are not printed because they take up a lot of space and therefore cause the overall graph to be shrunk to such an extent to fit on a page that the result is very hard to read.

-f *FSpec*
: FSpec is a function object specification. A function (loop, block) object specification can be either the symbol "+", indicating that all functions (loops, blocks) are to be displayed; or a comma-separated list of function (loop, block) object selectors. A function (loop, block) object selector is one of the following:

  an object name: specifies that object;

  wt = NNN or ct = NNN, where NNN is a floating point number between 0 and 1. This specifies all function (loop, block) objects with weight ("wt") or count ("ct") at least NNN of the total.

  NOTE: At this time, functions don't have counts associated with them.

  wt = :N or ct = :N, where N is an integer. This specifies the N function (loop, block) objects with maximal weight ("wt") or count ("ct").

-l LSpec LSpec is a loop object specification. Selecting a loop to be printed implies that the function it occurs in is also selected.

-b BSpec BSpec is a block object specification. Selecting a basic block to be printed implies that the function it occurs in is also selected.

-i fname fname is the input file. Default: stdin.

-o fname fname is the output file. Default: stdout

-s *fname* fname is the name of the file into which (the functions containing) the selected objects is printed in sloop source format. This option is provided because input files are often very large, and therefore time-consuming to read in, whereas often we'll be interested in only a few objects. Using the "-s" option, these can be selected and printed into a file in sloop source format, and this file can then be used as the input to sloop subsequently.

-d *fname* *fname* is the name of a file into which the dot specification of the output graph is to be written. The resulting file is not deleted at the end of processing. This option allows the user to modify the input to dot in ways not possible from the sloop command line.

-t "tile" – if specified, instructs the back-end graph-drawing tool to print large graphs on multiple pages (which can be "tiled" together to produce a single graph) instead of reducing it excessively in order to fit on a single page.

-c Print only basic blocks and edges on the critical path.

-C Print only basic blocks and edges on the critical path, as well as the instructions in them. Equivalent to -c -I.

The selector options -f, -l, and -b can be used together, e.g., the query

sloop -f -l wt = :3

specifies that the 3 heaviest loops should be printed out, together with the functions they occur in.

Options must be specified separately, each option preceded by a "-". E.g., the command "sloop -f -l -b 1000" specifies that basic block 1000 should be printed, together with the function and the loop it occurs in; but "sloop -fl -b 1000" asks for basic block 1000 and function "l".

2.3 Summarize Manpage

Introduction

summarize is a tool that takes the often lengthy output produced by alto and compresses it into a few lines

Invocation

summarize.perl <*alto-output*

3 Developing

3.1 Coding Guidelines

If you modify ALTO stick with the formatting and nameing conventions used in the code.

Every function is either static or extern. If it is extern please describe the function before its declaration. The description will be extracted into texinfo document if it is nested between '`@FUN`' and '`@END`'.

3.2 Compiling ALTO.

You need gnu-make (gmake) to compile alto. Make sure that the environment variable '`OSTYPE`' has the value '`osf1`' if you plattform is Digital Unix and some other value if you plattform is Linux.

No type '`gmake headers`' to produce some headerfiles. and the type '`gmake`' to produce the executable '`alto`'.

3.3 Plattform Issues

COFF Binaries

(A port to Linux ELF is under way !!) Currently, ALTO works with COFF binaries only. It assumes that those binaries are statically compiled. It also expects relocation information and symbol table information to be present.

Segements and Sections

A COFF binary consists of 3 segments: text, data, bss. Each segment consists of sections.

Code Sections

Not all section of the text segment contain code but only sections of the text segment contain code. ALTO merges the code containing sections into a single sections. It is helpful that those sections are adjacent in the text segment.

Readonly Sections

All sections in the text segment are readonly but there are sections in the data segment that are also readonly. It is necessary to know which sections are readonly so that loads from these sections can be "evaluated".

Relocation Information

ALTO uses relocation information to to determine basic block boundaries and to find basic blocks that can be the target of indirect jumps. ALTO will also "fix" relocatable data before the binary is written back.

Relocation Types

ALTO only considers relocation entries referencing the text segment. Luckily, there aren't all this many. There 8 byte addresses (REFQUAD) and 4 byte gp relative addresses (GPREL32) and one other class of relocation entries.

Refquads indicate the beginning of a basic block that can be jumped to from anywhere.

This is a rather coarse (but safe) assumption. Most of the time the refquad is used to describe a function address which is used in an non-computed jump, it would be nice to know which ones are used for computed jumps.

Gprel32s indicate the beginning of a basic block that is target of a switch statement (computed jump).

It is a big pain to figure out what the possible targets of a switch statement are and it would be nice if the symbol table provided this information.

There is a third type of relocation information used with init and fini sections.

Symbol Table

The symbol table is not used all this much. ALTO uses it to associate names with addresses (eg. function name) and to find function boundaries. The latter could also be achieved using procedure descriptors.

Procedure Descriptor table

Procedure descriptors are not used anymore but they contain important information and I might look into them again.

Useful Information That Would Be Nice To Have

Some of this information is approximated in ALTO

- targets of computed jumps
- Save/Restored register in a function
- Save/Restored register around a function call
- List of functions whose address is taken (in source or implicitly) which might hence be the target of computed jumps.

3.4 Concurrent Versions System

As more people work on ALTO (at the UofA) we will need to switch to a less restrictive versioning mechanism. CVS seems to be a good candidate.

3.5 Revision Control Systems

Setting Up the Development Environment for Alto

This is only relevant for shared delopment (at the UofA) Currently, we are using rcs but we are planning on swiching to cvs soon.

'`mkdir src`'
: Create a **local** directory that will hold a **shadow** copy of the **master** source directory.

'`cd src`'
: Change into the newly created directory.

'`ln -s /r/sin/home/debray/ALTO/src Master`'
: Add a symbolic link named '`Master`' in your **shadow** directory to the master directory. This make access to the master easier and is needed by the '`fix.csh`' script.

'`lndir Master`'
: Initialize the **local** (**shadow**) directory. This will fill the **shadow** directory with symbolic links to the **master** source files. The idea is to have symbolic links (shadows) to those files that you HAVE NOT changed and local copies of the files you HAVE changed. Ie. others will not see your changes until you want them to.

'`ln -s Master/RCS RCS`'
: The above did, unfortunately, not achieve a link to the RCS directory. You have to do this manually. If you forget to do this it seems that changes you make will be immediatelly visible to everybody right away. This is bad!

'`gmake headers`'
: Create header files

'`gmake`'
: This will produce the ALTO binary.

Checking Out a File

To check out a file you load the file from your **shadow** directory (NOT the **master** directory!) into Emacs. You then check it out using *C-x C-q* or the menu option '`Tools:Version Control:Check Out`'.

This will make the file writable for you and the corresponding file in the **master** directory is now "locked" for everyone else (pure magic!). They can still read the original version but they cannot change it.

If you now make changes and save the file, the new file will be saved in your **shadow** directory and not in the **master** directory. Everyone else will still see the (locked) original.

Checking In a File

When you have updated a file and made sure that it still works, you want to put it back in the master directory for everyone else to see. You do this i two steps.

First you check in the file in Emacs using *C-x C-q* again. This gives you the opportunity to write a comment on your changes. Type *C-c C-c* when your comment is written. The file in the **master** RCS directory is now updated. No update of the **work** file in the **master** directory itself taken place. To make this happen you must use the shell script '`fix.csh`' in your **shadow** directory. Run it in your **shadow** directory with the name of the file you just checked in as the only argument.

'`./fix.csh` *file*'

This script also replaces the file in your directory with a symbolic link to the file in the **master** directory again. This will make future changes made by others visible to you.

Creating a Snapshot of the Lastest Checked in Version

Change into the main src directory.

' `rcs -n` *snapshot-name*`: *`'

Checking Out a Snapshot

'`co -r`*snapshot-name* `src/RCS/*`'

Stealing a Lock

'`rcs -u` *file* '

3.6 Testing Alto for Speed and Correctness

Testing A New Alto Execuatble for Correctness

Create a local directory holding the files generated by the test scripts.

set the environment variable "altotestdir" to this directory. Do not use relative paths! Avoid a trailing slash ("/") in the path!

'`setenv altotestdir ~user/alto/testdir`'

set the environment variable "altotestexe" to the alto binary you want to check. Do not use relative paths!

'`setenv altotestexe ~user/alto/new_alto`'

You can also specify some arguments for alto but then you have to use quotes around the file and the options.

'`setenv altotestexe "~user/alto/new_alto -O loop"`'

Run the test script

'`/r/sin/home/debray/ALTO/bin/check.tcsh`'

The files generated in the "altotestdir" directory are as follows (since they are often very big, the script might delete some of them):

".log" alto output during optimization

".exe" the optimized executable

".out" output of optimized executable on test data

".dif" diff with correct output

Testing A New Alto Execuatble for Speed

Testing alto for speed improvement works similar to the correctness test. Only a different script is used:

'`/r/sin/home/debray/ALTO/bin/speed.tcsh`'

Instead of comparing the timings wiht the "unprocessed" versions of the programs you can specify a second alto binary in the environment variable "altotestexe2" to generate programs with.

'`setenv 'altotestexe2 ~user/alto/new_alto -O loop'`'

This enables you to either compare different versions of alto, or the same version but with different command line options (turning particular optimizations on or off for instance.)

Running Alto without Checking

'`/r/sin/home/debray/ALTO/bin/nocheck.tcsh`'

Running Alto to get performance counter measurements

'`/r/sin/home/debray/ALTO/bin/scrutinize.tcsh`'

This script has not been updated recently so it might be broken. Please check ftp://ftp.cs.arizona.edu/people/muth/pfm.tar.gz for the latest version of the pfm tool which is used by scrutinize.

Testing Only A Subset Of The Benchmarks

You can check only a subset of the benchmark suite by giving an extra argument to "check.tcsh" or speed.tcsh.

The argument is interpreted as a file name. Only those benchmarks will be used that have a file whose name matches that argument in their subdirectory.

The benchmarks reside in "/r/sin/home/debray/ALTO/benchmark".

Supose you have trouble with the "go" and xlisp benchmark. Do

'`touch /r/sin/home/debray/ALTO/benchmark/go/badbadbad`'

'`touch /r/sin/home/debray/ALTO/benchmark/xlisp/badbadbad`'

Now you can use

'`/r/sin/home/debray/ALTO/bin/check.tcsh badbadbad`'

to just check those two benchmarks.

In order to figure out what subsets have been defined already use:

'`/r/sin/home/debray/ALTO/bin/info.tcsh`'

4 Datastructures

4.1 Datastructure Intro

The alpha is a 64 bit architecture. Therefore, the use of pointers is rather expensive. In alto all objects are accessed by a 32 bit index relative to a base address rather that directly by a 64 bit address, ie. the objects are stored in an array. The drawbacks is are that (1) one has to know in advance how many objects of a certain type will be used during program execution and that (2) accessing objects is a little cumbersome.

We assume rather high upper bounds on the number of objects to remedy the first problem. They can also be changed using comand line options. Since the OS uses demand paging that is allocated memory does not really consume RAM unless it is used there is no big disadvantage. We address the second problem by providing macros to access object members.

Having all objects of a certain type in an array increases locality and allows us to iterate over them without an additional data structure.

The elements of an object are accessed indirectly using macros this allows us to change the underlying datastructures with having to change the entire program, eg. one of the plans is to stripe the individual components of a datastructure. In fact if one needs additional elements of an object for one particular optimization we suggest to allocate a separate array just for these new elements and define the necessary macros. An example can be found in "eval.c".

Since alto is rather space efficient we can affort the luxury of never deleting an object in order to reuse its space. Instead we just allocate a new object. This eliminates bugs to dangling pointers and simplifies iterating over lists which are simultaneously modified. New objects will always have higher numbers than old ones!

4.2 Function Datastructure

For each function a FUN object is allocated.

A function consists of a doubly linked list of basic blocks. The first bbl of this list is the init bbl and the last bbl is the exit bbl

4.2.1 Function Types

4.2.2 Function Members

`AFUN_FLAGS`
: FUN flags

`AFUN_BBL_FIRST`
: first BBL in the doubly linked list of BBLs belonging to this FUN

`AFUN_BBL_LAST`
: last BBL the doubly linked list of BBLs belonging to this FUN

`AFUN_NAME(i)`
name of FUN

`AFUN_STK_MAX`
set Stack Analysis

`AFUN_STK_MIN`
set by Stack Analysis, normally contains the (negative) framesize of the function.

`AFUN_REGS_USED`
set by the Liveness Analysis. Registers which will be subsequently used (not necessarily in this FUN, though.)

`AFUN_REGS_THROUGH`
set by the Liveness Analysis. Registers which are live at function entry if they are life at function exit.

`AFUN_REGS_SAVED`
set by the Saved Changed Analysis. Register Saved by this FUN.

`AFUN_REGS_CHANGED`
set by the Saved Changed Analysis. Register Changed by this FUN.

`AFUN_IS_RECURSIVE`

4.2.3 Function Iterators

`FOREACH_FUN(f)`
iterate (forward) over all FUNs

`FOREACH_LIVE_FUN(f)`
iterate (forward) over all not killed FUNs

`FOREACH_LIVE_FUN_R(f)`
iterate (backward) over all not killed FUNs. Iterating backwards often prevent infinite loops when new object are allocated during the iteration.

4.2.4 Function Iterators

4.2.5 Function Flags

`FF_STK_INVALID`
set by Stack Analysis. FUN behaves really bad stackwise.

`FF_STK_ARG`
set by Stack Analysis. FUN has stack arguments.

`FF_STK_ALIAS`
set by Stack Analysis. FUN has stack aliases

`FF_STK_BAD_ALIAS`
set by Stack Analysis. FUN has bad stack aliases

`FF_SETJMP`
: set by SpecialSetJmpLongJmp. FUN is a setjmp style FUN.

`FF_LONGJMP`
: set by SpecialSetJmpLongJmp. FUN is a longjmp style FUN.

`FF_PSEUDO`
: FUN is not a real FUN.

4.3 Basic Block Datastructure

For each basic block a BBL object is allocated. BBLs belonging to one FUN are connected in a doubly linked list.

4.3.1 Basic Block Types

`BT_CALL`
: a call BBL (bsr,jsr). Has two outgoing edges, ABBL_EDG_CALL and ABBL_EDG_LINK

`BT_JUMP`
: a jump BBL (jmp) Has one outgoing edge to HELL

`BT_SWITCH`
: orginally a BT_JUMP BBL but we could determine the possible jump targets. Has one outgoing edge for each possible target.

`BT_NORM`
: a BBL that branches unconditonally to some other BBL Has one outgoing edge

`BT_UNCOND`
: orginally a BT_NORM but code layout has added a br instruction Has one outgoing edge

`BT_BRANCH`
: a BBL with a conditional branch (IT_IBR or IT_FBR). Has two outgoing edge, ABBL_EDG_FALSE and ABBL_EDG_TRUE

`BT_EXTRA`
: a BBL that branches unconditonally to some other BBL in order to do some address computation. we try to eliminate all of these!

`BT_RET`
: a BBL that branches unconditonally to some other BBL Has one outgoing edge to an exit BBL.

`BT_HALT`
: a BBL with pal halt instruction. No outgoing edge.

`BT_EXIT`
: Exit BBL must be last in each FUN. Has outgoing edges to all the returnsites.

`BT_PAL`
: a BBL with pal instructions but not pal halt. Handles like a call to a pseudo function.

4.3.2 Basic Block Members

`ABBL_FLAGS`
: flags of this BBL

`ABBL_FUN`
: FUN this BBL belongs to

ABBL_NEXT
: next BBL in the doubly linked list of BBLs belonging to a FUN.

ABBL_PREV
: previous BBL in the doubly linked list of BBLs belonging to a FUN.

ABBL_PRED
: first EDG in the single linked list of incoming EDGs

ABBL_SUCC
: first EDG in the single linked list of outgoing EDGs

ABBL_INS_FIRST
: first INS in the doubly linked list of INSs belonging to this BBL

ABBL_INS_LAST
: last INS the doubly linked list of INSs belonging to this BBL

4.3.3 Basic Block Iterators

FOREACH_BBL(i)
: iterate (forward) over all BBLs

FOREACH_FUN_BBL(f,b)
: iterate (forward) over all BBLs b in a given FUN f

FOREACH_FUN_BBL_R(f,b)
: iterate (backwards) over all BBLs b in a given FUN f

4.3.4 Basic Block Functions

4.3.5 Basic Block Flags

FB_ALIGNED
: The BBL layout routine has determined that this BBL should be aligned because it might a frequent target of jumps.

FB_CRIT_PATH
: When we pathinline a function all inlined basic blocks should be treated as if they are part of the critical path for subsequent inlines. This is what this flag achieves.

FB_CRITICAL
: The basic block is crtical, ie. it is either hot connects hot basic blocks with the function init or exit node.

FB_SCHEDULED
: Used by the scheduler (should not be visible)

FB_LOOP_HEADER
: Basic block is loopheader. Set by the Loopanalysis routine.

`FB_CRIT_ESC`
: Basic block is a callsite leaving the inlined path

`FB_PROG_ENTRY`
: The bbl is the program entry point.

4.4 Instruction Datastructure

For each instruction a INS object is allocated. Instructions belonging to one BBL are connected in a doubly linked list.

4.4.1 Instruction Types

The instruction type is derrived from the opcode. The types are choose to simplify optimizations and scheduling.

`IT_MSC` trapb,excb,rpcc,amask,implver

`IT_LDA` lda, ldah

`IT_ILD` ldl,ldq,lduw,ldub

`IT_FLD` ldf,lds,ldg,ldt

`IT_ULD` ldq_u

`IT_LLD` ldl_l,ldq_l

`IT_IST` stq,stl,stw,stb

`IT_FST` stf,sts,stg,stt

`IT_UST` stq_u

`IT_CST` stq_c,stl_c

`IT_IBR` blbc,beq,blt,ble,blbs,bne,bge,bgt

`IT_FBR` fbeq,fblt,fble,fbne,fbge,fbgt

`IT_BRA` bra,bsr

`IT_JMP` jmp,jsr,ret,jsr_co

`IT_ICM` cmovlbc,cmoveq,cmovlt,cmovle,cmovlbs,cmovne,cmovge,cmovgt

`IT_FCM` fcmoveq,fcmovlt,fcmovle,fcmovne,fcmovge,fcmovgt

`IT_ICP` cmpbge,cmpult,cmpeq,cmpule,cmplt,cmple

`IT_FCP` cmpt?? cmpg??

`IT_FM4` muls?,mulf?

`IT_FM8` mult?,mulg?

`IT_FD4` divs?,divf?

`IT_FD8` divt?,divg?

`IT_F2I` ftois,ftoit

`IT_I2F` itofs,itoff,itoft

`IT_LOG` and,bic,bis,ornot,xor,eqv

`IT_SFT` msk??,ins??,ext??,zap??,s??

`IT_MLL` mull,mull/v

`IT_MLQ` mulq,mulq/v

`IT_MLH` umulh

`IT_IOP` rest

`IT_FOP` rest

`IT_RX` rc,rs

`IT_FCY` cpys

`IT_MEM` mb,wmb,fetch,fetch_m

`IT_EXT` sextb,sextw

`IT_PAL` call_pal

`IT_F2R` mt_fpcr

`IT_R2F` mf_fpcr

`IT_ERR` error condition

4.4.2 Insruction Members

`AINS_FLAGS`
flags of this INS

`AINS_BBL` BBL this INS belongs to

`AINS_NEXT`
next INS in the doubly linked list of INSs belonging to a BBL.

`AINS_PREV`
previous INS in the doubly linked list of INSs belonging to a BBL.

4.4.3 Insruction Iterators

`FOREACH_INS(i)`
iterate (forward) over all INSs

`FOREACH_BBL_INS(f,b)`
iterate (forward) over all INSs i in a given BBL b

`FOREACH_BBL_INS_R(f,b)`
iterate (backwards) over all INSs i in a given BBL b

4.4.4 Insruction Functions

4.4.5 Insruction Flags

4.4.6 Miscellaneous

Alto uses the DEC Alpha machine instructions as its intermediate code. Therefore, it inherits all its peculiarities.

Defined and Used Registers

A typical Alpha instruction reads (uses) two registers and writes (defines) a third one.

In ALTO you can access those registers using the macros AINS_REG_A(iins), AINS_REG_B(iins) and AINS_REG_C(iins) where "iins" is an instruction id. The macros return an 8-bit-signed number (type REG) specifying the register. 0-31 represents an integer register, 32-63 represents a floating-point register, 64 represents an integer intermediate value. (The file "regs.h" contains the corresponfing symbolic definitions.) When an instruction does not make use of all three registers the unused ones contain either 31 or 63 which are the integer/floatin-point zero registers. Two instruction subgroups do not adhere to this convention:

conditional-move-instructions (IT_ICM,IT_FCM) implicitly read the register AINS_REG_C(iins) besides writing it.

In order to shield the programer form these exceptions two functions have been introduced to ALTO:

AinsDef(iins) returns a 64-bit bit-mask (type REGLIST) representing the read registers of the instruction. AinsUSe(iins) returns a 64-bit bit-mask representing the written registers of the instruction. Here bit number i in the mask represent register number i.

The shielding is not perfect though since a conditional-move-instructions does not always define its target register.

4.5 Edge Datastructure

For each edge in the control flow graph an EDG object is allocated. Edges that are successors of one BBL are connected in a single linked list. Edges that are preddecessors of one BBL are connected in a single linked list.

4.5.1 Edge Types

<dt> ET_BR_TARGET <dd> Used With: JMP and BR.
 Head: BBL containing jsr/bsr instruction. Tail: Init BBL of target function.
 Stored in: ABBL_BR_TARGET

<dt> ET_FALL_THRU <dd> Used With: Fall Throughs
 Head: this BBL
 Tail: Next BBL
 Stored in: ABBL_FALL_THRU

<dt> ET_EXIT_LINK <dd> Used With: RET
 Head: BBL containing RET instruction, BBL ABBL_JMP (0)
 Tail: Exit BBL (dummy), relocatable BBL

<dt> ET_SWITCH <dd> Used With: RET
 Head: BBL containing RET instruction, BBL ABBL_JMP (0)
 Tail: Exit BBL (dummy), relocatable BBL

<dt> ET_BR_DUMMY <dd> Used With: RELOC and setjmp/longjmp type functions
 Head: BBL containing RET instruction, BBL ABBL_JMP (0)
 Tail: Exit BBL (dummy), relocatable BBL

<dt> ET_COPENSATE <dd> Used With: RELOC bbls
 Head: BBL containing RET instruction, BBL ABBL_JMP (0)
 Tail: Exit BBL (dummy), relocatable BBL
 </dl>

`ET_FUN_CALL`

Used bbls whose last ins is jsr,bsr or pal_call (BT_CALL,BT_PAL).

Head: bbl containing jsr/bsr/pal_call instruction.

Tail: Init BBL of target function. This will be the init bbl of

PSEUDO_HELL in case of a jsr and PSEUDO_PAL in case of a pall call.

Stored in: ABBL_EDG_CALL

ET_FUN_RETURN edges are automatically created/linked when ET_FUN_CALL edges are created/linked their index is on higher than the corresponding ET_FUN_CALL edge.

It is necessary that a ET_FUN_LINK edge be present when an ET_FUN_CALL is linked.

`ET_JUMP`

Used bbls whose last ins is a jmp whose targets could NOT be determined (BT_JUMP).

Head: bbl containing jmp instruction.

Tail: init bbl of PSEUDO_HELL

Stored in: nowhere

`ET_TRUE`

Used bbls whose last ins is a conditional branch (IT_IBR,IT_FBR) (BT_BRANCH).

Head: bbl containing conditional branch instruction.

Tail: condition true target

Stored in: ABBL_EDG_TRUE

`ET_FALSE`

Used bbls whose last ins is a conditional branch (IT_IBR,IT_FBR) (BT_BRANCH).

Head: bbl containing conditional branch instruction.

Tail: condition true target

Stored in: ABBL_EDG_FALSE

`ET_UNCOND`

Used bbls whose last ins is an unconditional branch (br) (BT_UNCOND).

Head: bbl containing unconditional branch instruction.

Tail: target

Stored in: ABBL_EDG_FALSE

Please note that ALTO deletes unconditional branches an reintroduces them during bbl layout. The bbl type is converted to BT_NORM and the edge type to ET_NORM.

`ET_NORM`

Used bbls which unconditionally branch to some target (BT_UNCOND).

Head:

Tail: target

stored in:

`ET_FUN_RETURN`

Head: exit bbl of a function

Tail: return site

stored in: nowhere because there might be a lot of them

ET_FUN_RETURN edges are automatically created/linked when ET_FUN_CALL edges are created/linked their index is on higher than the corresponding ET_FUN_CALL edge.

It is necessary that a ET_FUN_LINK edge be present when an ET_FUN_CALL is linked.

`ET_SWITCH`

Used bbls whose last ins is a jmp whose targets could be determined (BT_SWITCH).

Head: bbl containing jmp instruction.

Tail: target bbl (These are expected to be GPREL32 relocated bbls)

Stored in: nowhere since there might be a lot of them

`ET_FUN_LINK`

Used bbls whose last ins is jsr,bsr or pal_call (BT_CALL,BT_PAL).

Head: bbl containing jsr/bsr/pal_call instruction.

Tail: Next bbl (where control return after the function call)

Stored in: ABBL_EDG_LINK

`ET_HELL`

Used to model unknow controlflow.

Head: exit bbl of PSEUDO_HELL

Tail: Relocatable bbl (this is not very nice if relocatable bbl happens to be function init block, in such a case PSEUDO should CALL the is function)

Stored in:

`ET_EXIT_LINK`

Used with bbls whose last ins is RET (BT_RET)

Head: BBL containing ret instruction.

Tail: Exit BBL (dummy) of containing fun.

It is not necessary to specify the exit bbl when calling AedgCreateFancy. The routine will figure this out for you.

`ET_SETLONGJMP`
: Used to model spurious controlflow introduced by setjmp/longjmp style functions

`ET_COMPENSATE`
: Used to compensate for interprocedural controlflow other than calls.

`ET_EXTRA`
: Used with branch instructions that generate known addresses by jumping to the next instruction. Soon there will be no need for this anymore. The only remaining occurrence of such a branch is for setting the initial gp.

4.5.2 Edge Members

`AEDG_FLAGS`
: flags of this EDG

`AEDG_HEAD`
: = AEDG_FROM

`AEDG_FROM`
: BBL this EDG is pointing from

`AEDG_TAIL`
: = AEDG_TO

`AEDG_TO`
: BBL this EDG is pointing to

`AEDG_SUCC`
: next EDG in the single linked list of EDGs pointing from AEDG_HEAD

`AEDG_PRED`
: next EDG in the single linked list of EDGs pointing to AEDG_TAIL

The AEDG_FROM and AEDG_TO were originally called AEDG_HEAD and AEDG_TAIL but since they were defined in the opposite way from the Dragon book their use is not encouraged (although they still exists):

4.5.3 Edge Iterators

`FOREACH_EDG(e)`
: iterate (forward) over all EDGs

`FOREACH_BBL_SUCC(b,e)`
: iterate (forward) over all EDGs e pointing from BBL b

`FOREACH_BBL_PRED(b,e)`
: iterate (forward) over all EDGs e pointing to BBL b

4.5.4 Edge Functions

4.5.5 Edge Flags

4.5.6 Miscellaneous

Link, Call and Return edges

Given:

ibbl callsite next returnsite target entry node of callee fun

task: insert next_new as new returnsite

Code:

```
INDEX next_new = AbblNew( );
ABBL_TYPE(next_new) = BT_NORM;
ABBL_FUN(next_new) = ifun;
AbblAppendFancy(next_new);
AedgLinkFancy(AedgNewFancy(next_new,next,ET_NORM));

AedgKillFancy( ABBL_EDG_CALL(ibbl));
AedgKillFancy( ABBL_EDG_LINK(ibbl));

AedgLinkFancy(AedgNewFancy(ibbl,next_new,ET_FUN_LINK));
AedgLinkFancy( AedgNewFancy(ibbl,target,ET_FUN_CALL));
```

5 Control Flow Graph

5.1 Introduction

Great effort has been made to make the control-flow-graphs (CFGs) produced by ALTO as precise and useful as possible. Precise means all possible controlflow is modelled. Useful means the design of interprocedural analyses is simplified. The preciseness requirement lead to the introduction of a variety of edges that might be confusing at first.

The general flavor is based on interprocedural control-flow- graphs (cite something). So a function call is modelled using three edges:

call edge
: connects the callsite site with init bbl in the callee (first bbl)

return edge
: connects the exit bbl in the callee (last bbl) with the return site

link edge
: connects the callsite with the corresponding return site

5.2 Pseudo-Functions

A number of pseudo functions are added to model unknown controlflow.

PSEUDO-HELL

PSEUDO-PAL

We plan on adding a additional Pseudo function for setjmp longjmp

5.3 Compensation Edges

Each edge that crosses procedures boudaries has an associate edge wich models the control flow back to the function. If i is the id of the interprocedural edge than i+1 is the id of the associated edge.
 For a call edge (ET_FUN_CALL), the associated edge is a return edge (ET_FUN_RETURN) and leads from the exit bbl of the called function to the bbl after the callsite.
 For all other edges, the associated edge is a compensation edge (ET_COPENSATE) and leads from the exit bbl of the called to the exit bbl of the calling function.
 The existance of interprocedural edges that are not call edges is due to handcoded assembly langugage in some of the libraries.

5.4 Setjmp/Longjmp Type Functions

For some functions the ususual call and return edges are not enough to model the control flow accurately. For setjmp type functions an extra edge of type ET_SETLONGJMP is added from AFUN_JSR to the exit bbl of the function. For longjmp type functions an extra edge of type ET_SETLONGJMP is added from the exit bbl of the function to AFUN_JSR.

5.5 Relocation Information

Relocation information is needed to indentify possible target of computed jumps like jmp,jsr.

If the target address of a jmp or jsr instruction can be computed the instruction will be "strength reduced" to its non-computed counterpart jmp->br,jsr->bsr:

This conversion in done as part of the "partial evaluater optimization", so the CFG might change over time.

If this strength reduction is not possible we assume that all relocatable bbls are possibe targets (via PSEUDO_HELL).

We exploit the fact that there are different types of relocation entries:

gprel32 32 bit offset from the gp register, used for switch statements and function calls.

refquad 64 bit absolut, used for function calls and nonstandard table jumps

6 Execution Phases

Loading

very easy optimiations
special transformations
determination of BBL boundaries
construction of CFG

Construct Flowgraph and Datastructures (`BinaryConstructFlowgraph`)

- Increase size of date section to make space for new jumptables
- Mark function beginnings
- Mark basic block beginnings
- Use symboltable and reloc information to mark even more
- Create basic blocks and functions
- Create edges

Optimize Binary (`BinaryProcess`)

- Iteratively apply easy optimizations
- Hard optimizations (Inlining,Stackmerging)
- Iteratively apply easy optimizations

Assemble (`BinaryAssemble`)

- Layout all the basic blocks in the program in "a good order" and put them in a single function. Create unconditional branches where necessary.
- Determine which basic blocks should be aligned.
- Schedule basic blocks.
- Relocate the code

Saving

7 Functions

`extern void Usage()`

Print Alto usage information

`extern void ParseCommandLine(int argc, char *argv[])`

Parse command line and set internal flags accordingly. Allocate space for main datastructures (the size of which should evetually be configurable by commandline options)

`extern NUMBER AfunRegisterSaveOffset(INDEX ifun, REG reg)`

Assuming register reg is a saved register of ifun find its stack offset. NEEDS WORK!

`extern INDEX AfunRegSaveIns(INDEX ifun, REG reg)`

Assuming register reg is a saved register of ifun find its save instruction. NEEDS WORK!

`extern void AfunBypassSaveRegister(INDEX ifun,REG reg_saved, REG reg_bypass)`

Assuming register reg_saved is a saved register of ifun avoid the save and restore operations by bypassing it through reg_bypass

`extern void AfunKillLoadStoreSaveRegister(INDEX ifun,REG reg)`

Assuming reg is a saved register in ifun delete its save/restore pair.

`extern REGLIST AfunScratchRegisters(INDEX ifun)`

Find the registers in ifun which are neither used nore defined

`extern BOOL AfunIsLeaf(INDEX ifun)`

Return True if ifun does not call any other function.

`extern BOOL AfunIsLoopAnalyzable(INDEX ifun)`

Return True if ifun can be loop-optimized without making the optimizer go into an infinite loop

`extern NUMBER AfunReachableFirst(INDEX ifun)`

Return True if all bbls of ifun can be reached from the first (init) bbl

`extern NUMBER AfunReachableLast(INDEX ifun)`

Return True if all bbls of ifun can be (back) reached from the last (exit) bbl `extern NUMBER AfunNSwitches(INDEX ifun)`

Return the number of bbls of type BT_SWITCH in ifun

`extern NUMBER AfunNSwitchesCritical(INDEX ifun)`

Return the number of critical bbls of type BT_SWITCH in ifun

`extern NUMBER AfunNJumps(INDEX ifun)`

Return the number of bbls of type BT_JUMP in ifun

`extern NUMBER AfunNJumpsCritical(INDEX ifun)`

Return the number of critical bbls of type BT_JUMP in ifun

`extern BOOL AfunHasEsc2(INDEX ifun)`

Return True if ifun has outgoing escaping edges

`extern BOOL AfunHasEsc2Critical(INDEX ifun)`

Return True if ifun has outgoing escaping edges from critical bbls

`extern BOOL AfunHas2Esc(INDEX ifun)`

Return True if ifun has incoming escaping edges

`extern BOOL AfunHas2EscNotFirst(INDEX ifun)`

Return True if ifun has incoming escaping edges to bbls other than the init bbl

`extern BOOL AfunIsSetjmp(INDEX ifun)`

obsolete

`extern BOOL AfunIsLongjmp(INDEX ifun)`

obsolete

`extern REG AfunReturnRegister(INDEX ifun)`

Return the return register of ifun

`extern NUMBER AfunNCriticalCallSites(INDEX ifun)`

Return the number of call edges calling ifun from critical bbls, does include incoming escapes.

extern NUMBER AfunNHotCallSites(INDEX ifun)

Return the number of call edges calling ifun from hot bbls, does include incoming escapes.

extern NUMBER AfunNCallSites(INDEX ifun)

Return the number of call edges calling ifun, does include incoming escapes.

extern NUMBER AfunNInssCritical(INDEX ifun)

Return the number of critical inss in ifun

extern NUMBER AfunNInss(INDEX ifun)

Return the number of inss in ifun

extern NUMBER AfunNBblsCritical(INDEX ifun)

Return the number of critical bbls in ifun

extern NUMBER AfunNBblsHot(INDEX ifun)

Return the number of hot bbls in ifun

extern NUMBER AfunNBbls(INDEX ifun)

Return the number of bbls in ifun

extern void AfunKill(INDEX ifun)

Kill the function including it bbls edgs and ins.

extern INDEX AfunNewFancy()

Create a new essentially empty function.

extern void AbblBranchTwist(INDEX ibbl)

Assumning that ibbl has a conditional branch reverse the true and false branch.

extern BOOL OptBranchForwarding(FILE *fp)

Follow chains of empty basic blocks until a nonempty basic block is found which is returned.

extern INDEX AfunDupCriticalPath(INDEX ifun,double flow_frac)

Duplicate The Critical Path of a Function

extern INDEX AfunDup(INDEX ifun, double flow_frac)

Duplicate (Clone) a Function

extern BOOL OptDuplicateFunctions(FILE *fp)

Inline a function "callee" at the basic block "callsite"

extern BOOL AbblIsReachable(INDEX callfrom, INDEX callto)

Returns TRUE if it is possible to reach callto from callfrom via a sequence of function calls followed backwards, i.e., from callee to caller. This code is not very efficient, but can (and should) be made faster if it's found to be a bottleneck.

extern NUMBER AedgMarkRecursiveCalls()

extern BOOL AfunWillNotBeInlined(INDEX ifun,BOOL usepath)

extern INT32 AbblLoopNCriticalInss(INDEX loophdr)

extern INT32 AfunNCriticalInss(INDEX ifun)

extern BOOL AfunNotRecursivelyInlinable(INDEX callee, BOOL usepath)

Returns TRUE if function callee should not be considered for recursive inlining. At this time, a recursive function callee is not considered for inlining if any of the following hold:

(1) callee is a pseudo-function (shouldn't happen, but included for safety);

(2) callee contains a setjmp or longjmp;

(3) callee does not have a "good" callsite (hence, noweher for it to be inlined to);

(4) callee contains an indirect jump;

(5) callee contains a jump through a jump table (i.e., a switch);

(6) callee contains an "escaping" edge to another function;

If the boolean argument usepath is TRUE, only the critical path is considered.

extern REG RegsGetFirst(REGLIST rl)

Return the lowest positon of a set bit in rl.

extern BOOL AfunCallsBadFunction(INDEX ifun)

obsolete

extern BOOL OptimizeEasy(FILE *fp)

Iteratively Execute Easy Optimization This function calls all the easy optimizations.

extern BOOL OptimizeHard(FILE *fp)

The Hard Optimization. This function is called only once sandwiched between calls to OptimizeEasy. It calls all the optimizations that should be applied only once. Because of problems with jumptable an ugly call to OptimizeEasy was necessary.

8 Optimizations

`extern BOOL OptBranchTwist(FILE *fp)`

obsolete (is now part of layouter)

`extern BOOL OptBranchConditional(FILE *fp)`

If both branches of a conditional point to the same basic block the conditional is useless.

`extern BOOL OptBranchForwarding(FILE *fp)`

Forward branches over empty basic blocks.

`extern BOOL OptBranchMerge(FILE *fp)`

Merge basic blocks which are connected by a normal edge if the target basic block has no other incoming edges.

`extern BOOL OptBlockDuplicate(FILE *fp)`

currently disabled

`extern BOOL OptConstantsOne(FILE *fp)`

Compute constants faster.

`extern BOOL OptConstantsTwo(FILE *fp)`

Sophisticated constant propagation. We resuse a constant in some other register to (hopefully) reduce register usage. This is only done on the basic block level. We do not kill instructions but expect useless code to be eliminated by other optimizations. We also allow construction of values from other using either explict lda's or doing it implicitly for memory format instruction. Care must be taken to not construct values that point into code sections since those values may change!!!

`extern BOOL OptDuplicateFunctions(FILE *fp)`

Duplicate Functions. To reduce "bad data flow" into function called from unknown callsites, eg. "hell". All functions which have "normal" callsite and are called by hell are cloned (duplicated). The normal callsites will call one clone and hell will call the other.

`extern BOOL OptInline(FILE *fp,BOOL usepath )`

Inline Frequently Called Functions.

extern BOOL OptAvoidLoads(FILE *fp)

Avoid reloading of a value from memory. Pairs of store/load instruction that provably reference the same memory location are replace by a store/move pair.

extern BOOL OptCopyPropagation(FILE *fp)

Propagate copies. Uses of registers are replaced with uses of the oldest copy of that register. This optimization could be extended to make use of lda with nonzero offsets. To "freeze" the changes made by this optimization. Dead code should be removed right after it.

extern BOOL OptMoveElimination(FILE *fp)

Eliminate Moves

This optimization is a reasonable hack until we have a real register reallocator. The idea is that if we eliminate a move operation we

o save an instruction

o can potentially reduce the register usage

Given move x,y We have two options.

o we can make the defintion of x a defintion of y

o we can make all uses of y uses of x

The first option is explores here the second option is implicitly performed by the copy propagation elimination.

The interactions with the Copy Propagation Optimization are unclear.

Conditional moves are a pain in the butt and need very careful coding.

Once we have determined that optimization can be applied we need to do the following:

o the old defining instruction for x should now define y

o each use of x should be converted into a use of y once target becomes redefined we are done

The problems is with conditional move instructions which do not really define their target register:

o when we look for the defining instruction for x r we will ignore condtional moves.

Some bad cases:

```
def x
use x
use y (bad)
move x,y
```

```
def x
move x,y
use y
redef y
use x (bad)
```

the following code appears in cos

```
def x
move x,y
condmove a,x
condmove y,z  (bad)
```

extern BOOL OptStackSaveRegs(FILE *fp)

Avoid Unneccessary Callee Register Savings Not implemented yet but code is scanning for opportunities and prints them.

extern BOOL OptDeleteStack(FILE *fp)

Delete Unused Stack. A stackframe which is never used will be deleted. This optimization seems to be not very effective. and prints them.

extern BOOL OptLeafSaveRegisters(FILE *fp)

Avoid Callee Save Registers in Leaf Functions CURRENTLY NOT USED.

extern BOOL OptCalleeSaveRegisters(FILE *fp)

Delete unused stacks. Currently almost identical to OptStackSaveRegs.

extern BOOL OptStackMerge(FILE *fp)

Merge stacks. This optimization tries to reduce the number of stack pointer manipulations to one at the beginning of the entry bbl and one before each return instruction.

extern BOOL OptTableJumps(FILE *fp)

Identify possible targets in a BT_JUMP basic block. If all targets can be identifies the block changes its type to BT_SWITCH

extern BOOL OptGprel(FILE *fp)

If all BT_JUMP block in a function have been converted into BT_SWITCH blocks. We feel it is safe to eliminate edges from HELL to GPREL32 basic blocks in this function.

extern void AbblInstallNewJumpTable(INDEX ibbl)

Given a basic block of type BT_SWITCH generate a new jumptable for it in the enlarged data section. This useful when we duplicated this basic block and now need to use a different jumptable than the original.

extern BOOL OptNormalizeInstructions(FILE *fp)

Normalize Instructions. Often there are several instruction which achieve the same effect eg, lda r0,5(r1) or add r1,5,r0. In order to reduce the number of cases to be considered by other optimizations we normalize instructions. Those normlization should be (but aren't) undone before scheduling picking the instruction which has the nicer scheduling behaviour, eg. use bis r5,r5,r6 instead of lda r6,0(r5)

extern BOOL OptEliminateNoops(FILE *fp)

Eliminate No-ops

extern BOOL OptRemoveUnreachableCode(FILE *fp)

Eliminate Unreachable Code .

extern BOOL OptUnlinkEmptyBbls(FILE *fp)

Eliminate Empty Basic Blocks. Remove empty basic blocks which must be a bt_norm basic block by forwarding its incoming edges to the next bbl Except if the basic block is a dummy entry basic block

extern BOOL OptPeepHole(FILE *fp)

Peephole optimizations and strength reductions

extern BOOL OptCondMoveOne(FILE *fp)

Conditional move generation Conditional moves are introduced for the basic block on the left hand side

```
          O
          |\
          | O
          |/
          O
```

extern BOOL OptCondMoveTwo(FILE *fp)

Conditional move generation. Conditional moves are introduced for the basic blocks in the middle

```
          O
         / \
        O   O
         \ /
          O
```

9 Analyses

extern void ComputeStackInfo(FILE *fp)

Analyze stack size and usage. This anlysis determines the stack behaviour of all the functions. Upon completion

AFUN_MIN(fun) contains the maximum stackframe size used my the function (this is a negative number). AFUN_MAX(fun) if the stackframe size varies this is the smallest stack framesize encountered. (this is a negative number).

For each basic block in fun

ABBL_STACK_CHANGE_IN(bbl) contains the stackpointer relative to what it was at function entry. Stackmerging invalidates this field.

A variety of flags are set or cleared for functions:

AFUN_HAS_STK_ARG(fun) the function accesses the stack outside of its frame AFUN_HAS_STK_ALIAS(ifun) the function copies the value of the stackpointer plus an offset to another register (using lda) AFUN_HAS_STK_BAD_ALIAS(ifun) the function uses the value of the stack pointer with an instruction different from lda. AFUN_HAS_STK_INVALID the function does bad things with the stack

Concept Index