FlashPort

You too can
Go Native
to PowerPC Macintosh

S. Shastry

FlashPort

Introduction
FlashPort Product
FlashPort Technology
FlashPort Demonstration
Conclusion

Introduction

Software Migration Alternatives

- **Emulation**
- **Traditional hand porting**
- **Binary translation**

Software Emulation

+Precise equivalence

-Performance loss

-Difficult to evolve

Traditional hand porting

+Maximizes performance
+Easy to maintain and evolve
-Precise equivalence difficult to achieve
-Porting effort is substantial

Binary translation

+Excellent performance
+Precise equivalence
+Maintainable
+Evolvable by replacement
-Requires precise and complex analysis

Binary Translation

Translation of applications in binary form

No changes in source required

Very sophisticated tool

- Low level of abstraction
- Analysis of all program paths
- Accomodate a number of programming paradigms
- Accomodate ill-behaved programs

Binary Translation
Two Possible Approaches

1. **Open Ended Translation**
   - **Most of code is found and translated**
   - **Rest of code is found at run-time and emulated**
2. **Bounded Translation**
   - **All of the code is found and translated**

FlashPort Product

FlashPort for Macintosh

Platforms:

- **Source Platform -- 68K based Macintosh**
- **Target Platform -- PowerPC based Mac**

Translation Goal:

- **Translation of complete executable binary**
- **Translation of subprogram object code**
- **Mix translated and compiled code**

FlashPort for Macintosh

Programs written in

- 68K Assembly
- C, PASCAL or any procedural language
- Mix of Assembly and procedural language

Has a graphical user interface. Generates:

- Call Graph
- Flow Graph

FlashPort for Macintosh

Generates

- 68K assembly listing
- PowerPC Assembly Listing
- PowerPC Assembly source
- debugging information

Is Native

FlashPort for Macintosh

Enabling Technology for Apple's strategy of Macintosh Application Services on Unix platform

FlashPort:
Proven Product

ECHO LOGIC

Using FlashPort, we have translated:

- **System 7 Macintosh ROM**
- **System 7 System Files**
- **System 7 Process Manager**

FlashPort:
Proven Product

Using FlashPort, we have translated:

- **Commercial Applications:**
  - **WordPerfect**
  - **Stuffit**
  - **ArchiCAD**
  - **HyperCard**
  - **Vette!**
  - **AfterDark Modules**
  - **Apps from other major vendors**

FlashPort: Proven Product

For translations performed so far, we:

- took commercial of-the-shelf software
- had no access to the source code of the app
- did not require any source changes
- have translated millions of lines of code

FlashPort Identifies

Self-modifying code
Run-time generation of 68K code
Accessing code as data, moving code
Manipulation of A5 Jump table

FlashPort Identifies

- Potential wild branches
- Code in global data area
- Routines that cannot be translated to native calling conventions
- Possible logic flaws in the program

FlashPort Technology

Binary Translation
has to accommodate

Hardware Differences:

- **Machine instructions**
- **Instruction sizes**
- **Distance between code blocks**

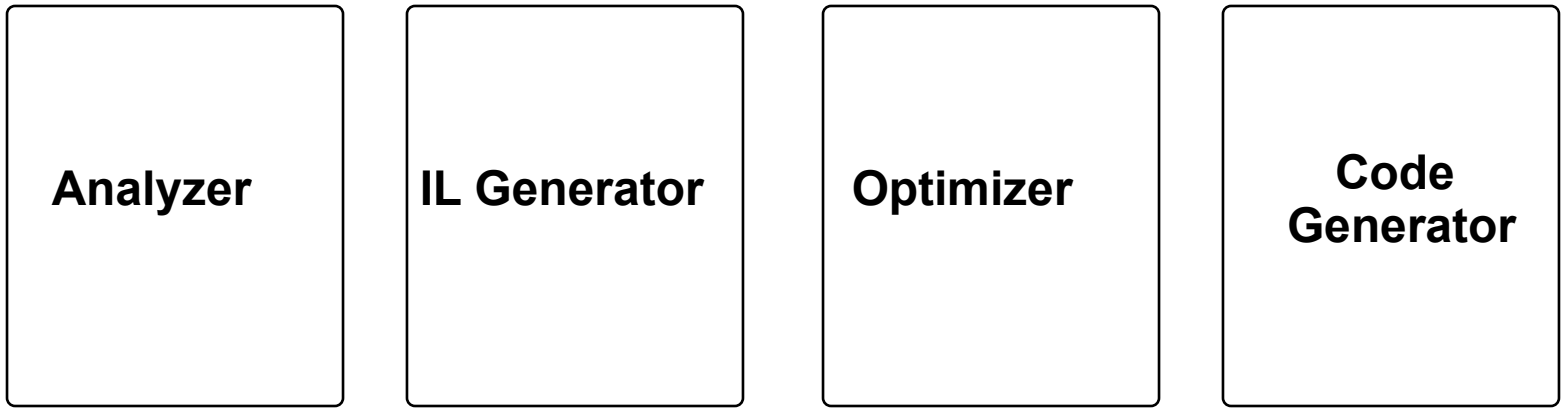
Binary Translation
has to accommodate

- OS/Run-Time model differences
  - Memory and stack layout
  - Subprogram calling conventions
  - Toolbox calls
  - Mixed mode calls

Overview of FlashPort

Front End

Back End



Figures out what program does and makes notes about how thing are used

Recasts program in machine-independent form

Globally optimizes machine-independent form

Binds to target machine performing machine-dependent optimizations

FlashPort Analyzer

- Follows flow of control from program entrypoint
- Divides program into procedures for analysis
- Alternates between Pass1 (simple analysis) and Pass2 (complex analysis) for maximal efficiency
- Derives (in Pass2) a picture of the machine state (called a context) for each block of a procedure
- Applies calling context of call site to exit contexts of callee (called arg-munging) to get effect of procedure call
- Uses Pass3 (analysis with actual arguments) where necessary

Example of Arg-Munging


```
proc1:  move.l  #3,a1
           pea     array
           jsr     proc2
retlbl: add.w   #4,sp
           ...
proc2:  move.l  4(a7),a0
           move.l  a1,(a0)
           rts
```

Placeholders for Pass2


```
REG %d0:   P%d0
...
REG %sp:   (stack + 4)
REG %pc:   retaddr
```

Exit Context for proc2


```
REG %a0:   *(long *)(stack[proc2] + 4)
REG %a1:   P%a1[proc2]
REG %sp:   stack[proc2] + 4
REG %pc:   retaddr[proc2]

BASE: stack[proc2]
   4: *(long *)(stack[proc2] + 4)
   0: retaddr[proc2]
BASE: *(long *)(stack[proc2] + 4)
   0: P%a1[proc2]
```

Calling Context for proc1


```
REG %a0:   P%a0[proc1]
REG %a1:   3
REG %sp:   stack[proc1] - 8
REG %pc:   proc2

BASE: stack[proc1]
   0:   retaddr[proc1]
   -4:  array
   -8:  retlbl
```

Results in proc1 after arg-munging


```
REG %a0:  array
REG %a1:  3
REG %sp:  stack[proc1] - 4
REG %pc:  retlbl

BASE: stack[proc1]
   0:  retaddr[proc1]
   -4: array
BASE: array
   0: 3
```

Deriving calling conventions

- a1 and 4(sp) are arguments to proc2 because they are used-before-set in proc2
- a0 may be a return from proc2 if it is subseqently used in proc1 before being reset
- Because it takes register arguments proc2 is classified as a non-standard routine
- However the calling sequence for proc2 is classified as normal because it is only jsr'ed to and it only has normal returns

FlashPort IL Generator

- Responsible for producing machine-independent version of program
- Extremely detailed and precise representation of instruction semantics
- Conversion of pointers to symbolic form
- Conversions of distances to symbolic form
- Biasing of pointer references
- Conversion of “code vector” jumptables

FlashPort: Optimizations

Dead code elimination
Unreachable code elimination
Constant propagation
Scalar propagation
Constant arithmetic
Symbolic arithmetic
Constant conditional evaluation

FlashPort: Optimizations

- Invariant code motion
- Register propagation
- Inverse scalar propagation
- Common subexpression elimination
- Range propagation

FlashPort: Optimizations

- Instruction reordering
- Register Allocation
- Pipeline scheduling
  - Load delay slot
  - Move from special registers
  - Across basic blocks

Automatic Generation of Native Code

- Convert ALINEs to Native Toolbox routine call
- Map stack-based parameter passing to register-based convention
- Create Routine Descriptors for Mixed-mode calls
- Use native floating point processor instructions
- Eliminate indirection through A5

FlashPort Demonstration

Conclusion

- FlashPort Binary translation is real
- Supports
  - Incremental evolution
  - Single code base
- Attractive alternative to hand porting
- Native App for PowerPC
- Reduces Time to Market