An Approach to Mixed Mode on Itanium under HP-UX

Grant Sidwall grant_sidwall@hp.com

Ken Kroeker retired

HP Enterprise Solution Partners

HP WORLD 2002 Conference & Expo

Agenda

- Platform transition technologies
- Dynamic Object Code Translation
- Two process Mixed Mode
- Shared Memory Mixed Mode
- RPC based Mixed Mode
- Results and Comparisons

Platform Transition

- The movement of an operational environment and/or application from one operating system and hardware architecture to another
- Done to achieve better price/performance
  - or to avoid obsolescence

Platform Transition Technologies

- Port/Recompile for new platform
  - Need ALL source and libraries (on new platform) - write portable code!
- Emulation - for programs with no source available, or too hard to port
- Dynamic Object Code Translation
  - the new ‘emulation’
  - still at the program level

Dynamic Object Code Translator

- What is DOCT?
  - PA-RISC to IPF Binary Emulator
- Supports 32-bit and 64-bit applications
- Proven reliability on many applications
  - Apache web server, NASTRAN, JAVA VM, Netscape Browser, WebDB, Spec95/2000, Xemacs (over 250 others)
- Internal code name - Aries

DOCT: binaries

- Four binaries:
  - shared libraries: aries32.so, aries64.so
  - loaders: pa_boot32.so, pa_boot64.so
- Reside in:
  - /usr/libhpux32/, /usr/lib/hpux64/
- loader(s) are invoked by the kernel loader
  - pa_boot32.so invokes aries32.so
  - pa_boot64.so invokes aries64.so

DOCT: specifying options

- All options to DOCT are specified using a resource file - .ariesrc
- DOCT search path for .ariesrc
  - System wide (/usr/local/aries/.ariesrc)
  - local to user ($HOME/.ariesrc)
  - current working directory ($PWD/.ariesrc)

DOCT: limitations (1)

- Only pure 32-bit or 64-bit applications are supported. No mixed mode is allowed, whether PA/IPF or PA32/PA64.
- Does not support code compiled on HP-UX 8.0 or earlier.
- Does not support privileged PA-RISC instructions. Hence, device drivers and loadable kernel modules are not supported.
- Does not support use of /dev/kmem - I.e.access to kernel structures

DOCT: limitations (2)

- Does not support timing dependent code, including applications that expect “real-time” response or assume that there is consistency in the amount of time it takes to execute a particular sequence of instructions.
- Does not support code that uses ptrace(), ttrace(), or profil() system calls (debuggers and profilers).
- Uses some of the process’ address space. ‘Large’ programs may run out of address space.

DOCT: limitations (3)

- Replaces vfork() with fork(). Applications that rely upon the difference between the two are not supported.
- Does not support PA programs that load IPF shared libraries.In other words, mixing PA binaries with IPF shared libraries is not supported. DOCT is meant only for pure PA binaries, i.e., binaries that are either statically or dynamically linked with PA libraries ONLY.

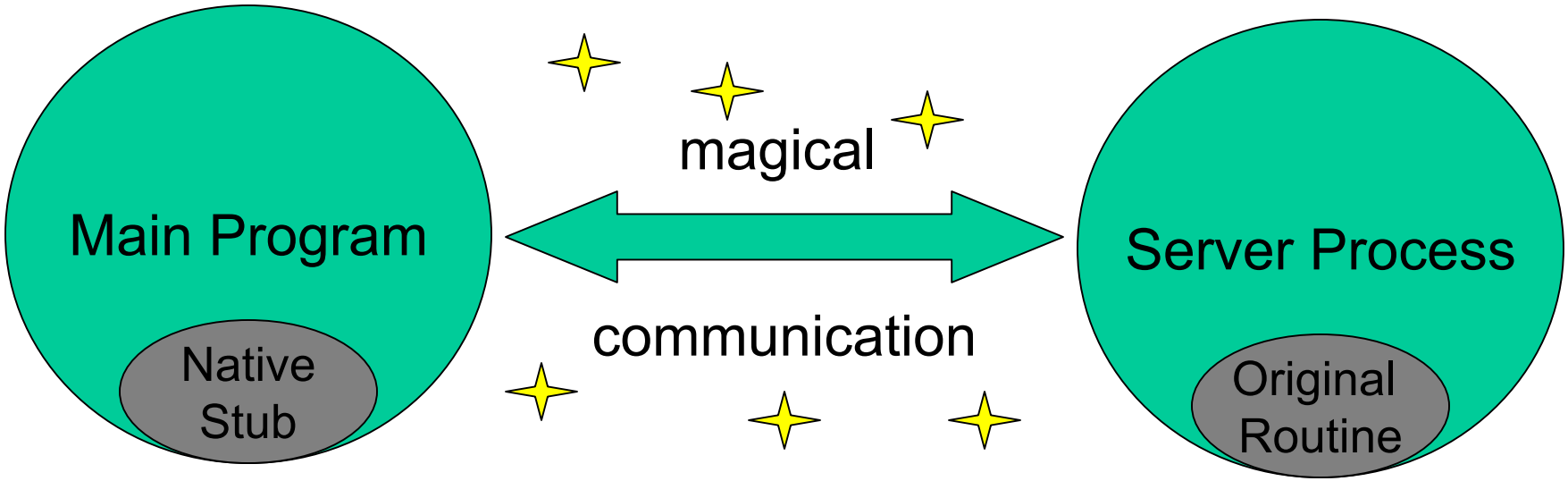
DOCT: Performance

- Compute intensive code relative to L-class:
  - Integer applications: 4x-5x slowdown
  - Floating point applications: 7x-10x slowdown
- Performance on user-interactive applications, eg. Netscape, Xemacs, Apache Web Server,etc.:
  - comparable to L-class
  - useable - no visible slowdown

Mixed Mode: who needs it?

- Pure DOCT is too slow
- Not enough time or resources to port/rewrite entire application, just key parts.
- Not all source code or libraries are available:
  - 3rd party library not ported to IPF yet
  - 3rd party library not ported to IPF ever
  - faster library available on IPF, but main not ported to IPF (interactive app with compute intensive library)

Mixed Mode: general architecture



The form of the inter-process communication is the major issue in making ‘two-process mixed mode’ work.

Forms of interprocess communication (ipc)

- Signals
- Shared files
- Pipes
- Message queues
- Semaphores
- Shared memory
- Remote Procedure Calls

Pros and Cons of Signals

- Pros
  - not hard to implement
- Cons
  - not suited to passing general data
  - predefined actions for many values
  - just not meant for this

Pros and Cons of Pipes

- Pros
  - can be bidirectional or used in pairs
- Cons
  - parsing / marshaling of data required at both ends
  - potential problems with binary data

Pros and Cons of Message Queues

- Pros
  - automatic synchronization of the two processes
- Cons
  - parsing / marshaling of data required at both ends

Pros and Cons of Semaphores

- Pros
  - meant for interprocess synchronization
- Cons
  - not meant for data transfer

Pros and Cons of Shared Memory

- Pros
  - low overhead - fast
- Cons
  - requires synchronization
  - have to be careful of 32/64 bit data alignment

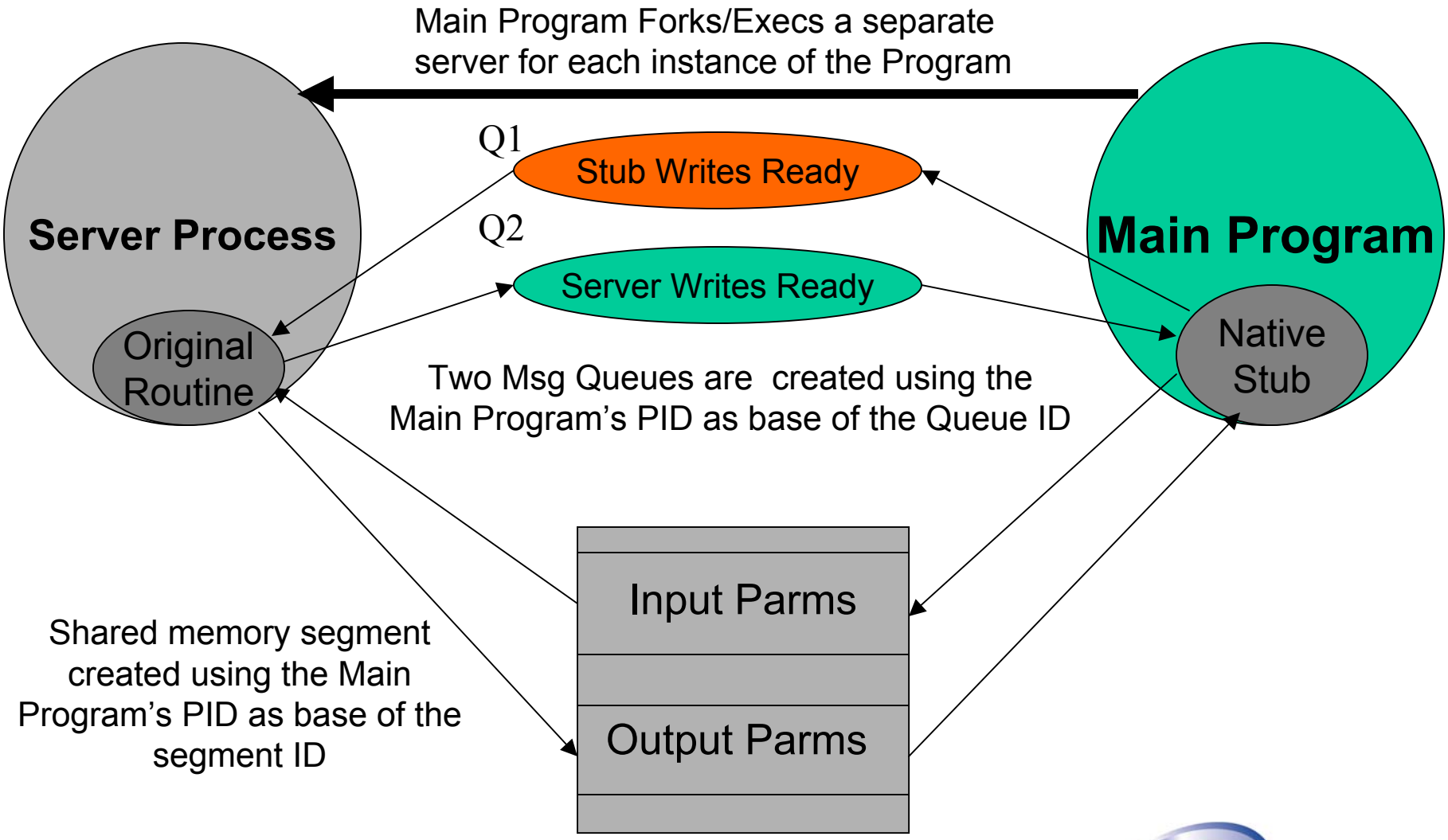
Pros and Cons of RPC's

- Pros
  - automatic data marshaling, conversion, parsing
  - allow processes to be on different architectures
  - allows use of code on PA where native PA performance is better than emulated code performance
- Cons
  - high overhead (slow)
  - not as widely used / well known

Topics for this Paper

- Shared memory with synchronization
  - fast execution - less data movement overhead
  - need to create the synchronization
- Remote Procedure Calls
  - based on DCE standard
  - platform independent - can use native code or emulation on host
  - no data marshaling or parsing
  - synchronization is inherent
  - needs some infrastructure (supplied in HP-UX 11i)

Mixed Mode Using Shared Memory



Pieces of the SM puzzle

- The shared memory solution has several pieces:
  - the (modified) original program that calls the library
  - the ‘stub’ routine that intercepts the library call, manages communication with the ‘server’, and returns the result
  - the ‘server’ program that manages communication and calls the library routine
  - the library that can’t migrate
  - the emulation environment (DOCT) that surrounds the non-native portion of the code (library or original program!)

Program changes - original program

- Declarations:
  - process ID variable for fork
- Before the library routine is called:
  - fork & exec the server program
- At the end of execution
  - kill the server process
- At link time:
  - load the stub routine instead of the library routine

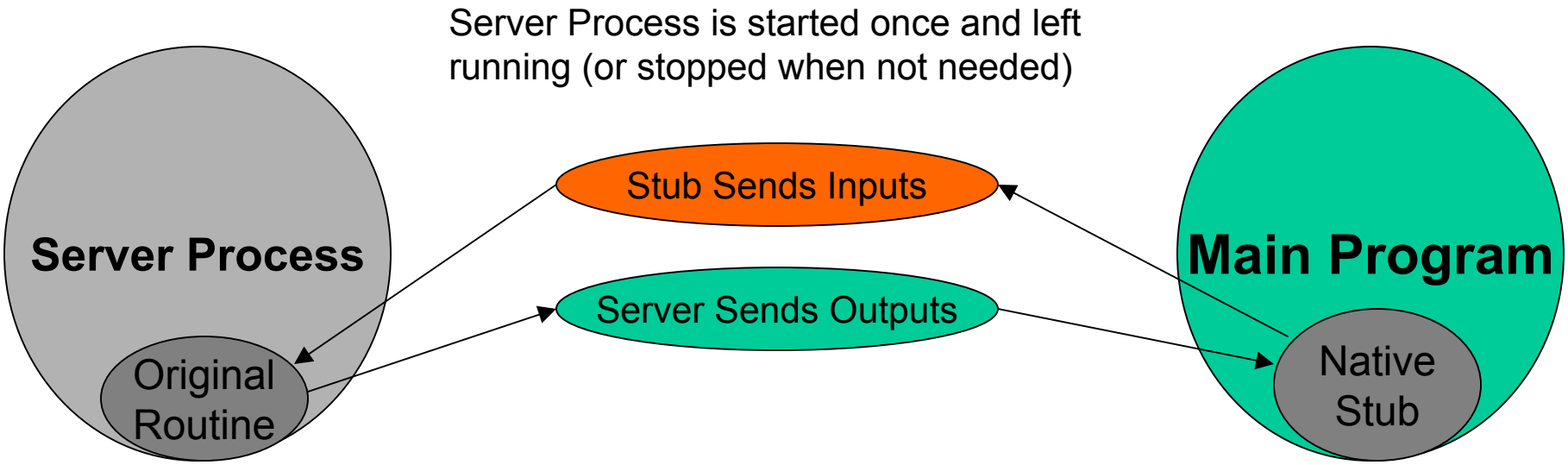
Program steps - intercepting stub

- Open the synchronization mechanism
- Wait for the server to create the shared memory segment, then attach to it.
- Put the input parameters into shared memory
- Signal the server that input is ready
- Wait for the server to signal that output is ready
- Save the output
- Detach the memory segment
- Close synchronization mechanism
- Return output as results

Program steps - legacy server

- Create the synchronization mechanism
- Create the shared memory segment; attach to it.
- Processing loop:
  - wait for input data to be placed in shared memory
  - retrieve the input data
  - call the library function
  - put the results into shared memory
  - signal that results are ready
- Start back at top of loop

Mixed Mode Using RPC



Establishment of communication is accomplished through the DCE based infrastructure in HP-UX, i.e. the server registers its existence, and the main program finds it when needed, and uses it.

Pieces of the RPC puzzle

- The Remote Procedure Call solution has quite a few pieces (template took time to create):
  - the RPC daemon (rpcd) that handles connections
  - the Makefile that builds the pieces
  - the idl file that describes the library's interface
  - the modified original program (client)
  - the 'container' routine that intercepts the library call, and returns the result
  - the 'server' program that manages communication and calls the container routine to get to the library routine
  - the library that can't migrate

Program changes - original program

- Declarations:
  - DCE, thread and idl include files, DCE connection handle
- At the start of execution
  - call function to establish DCE connection with server (function created from example code, not invented just for this - reasonably complex)
- Add onion skin to translate function call to remote RPC call
- At link time:
  - load the generated stub, DCE and thread libraries, add /opt/dce/lib to include file path

Program steps - container function

- Call the function and return the results

Program steps - server program

- Register the availability of this service with the rpcd daemon
  - this is complicated looking boiler-plate code
- Listen for remote procedure calls, and process them by calling the container function
  - this is just one call that does it all (the processing loop)
- Code to unregister the service - normally never used, as the server just waits for more calls until killed

Performance Comparison

- Simple program:
  - library routine increments argument and returns it
  - main loops, calling routine with arguments from 1 to upper bound, and adds up the results
  - main prints upper bound and final result
- Run on:
  - rp5450 (PA-RISC 440 Mhz - 4 cpu)
  - i2000a (IPF 800Mhz - 2 cpu)

Comparison Results

Time in seconds for x iterations:	10K	100K	1M
– SM - PA Native client and server	1.8	9.1	88.3
– SM - IPF Native client and server	2.0	10.9	100.6
– SM - IPF client, DOCT/PA server	2.0	11.1	102.2
– RPC - PA Native client and server	2.0	19.3	189.2
– RPC - IPF Native client and server	3.3	32.2	324.5
– RPC - PA client, IPF server, network	5.5	54.9	539.3
– RPC - IPF client, PA server, network	5.4	52.6	529.4
– RPC - IPF client, PA / DOCT server	9.3	87.3	842.6
– Native, monolithic code, IPF or PA	<0.1	<0.1	<0.1

Comparison Conclusions

- For large numbers of iterations, both methods scale linearly - call overhead becomes the main element in the time spent.
- For DOCT based activity (the most useful mode), shared memory is ~ 8 times faster.
- Fully compiled monolithic code is still orders of magnitude faster - measurable results < 1 sec at 10M iterations.
- RPC call overhead with DOCT is ~.9 msec
- SM call overhead with DOCT is ~ .01 msec

Comparison Conclusions

- So why would you use RPC?
  - Number of iterations is small
  - Data passed is complex, not easily aligned
  - When native PA performance is better than DOCT performance - keep code on PA, call via network
- And when would you use Shared Memory?
  - When performance is a consideration (ported code is still MUCH faster).
  - When data is fairly simple
  - Most of the time