HPCM: A Pre-compiler Aided Middleware for the Mobility of Legacy Code

Cong Du, Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
{ducong, sun}@iit.edu

Kasidit Chanchio
Computer Science and Mathematics Division
Oak Ridge National Laboratory
chanchiok@ornl.gov

Content

- Process migration motivations and overview
- Related research
- HPCM middleware architecture and its components
- The pre-compiler and its functionalities
- Experimental testings and results
- Conclusions and future work

Motivation

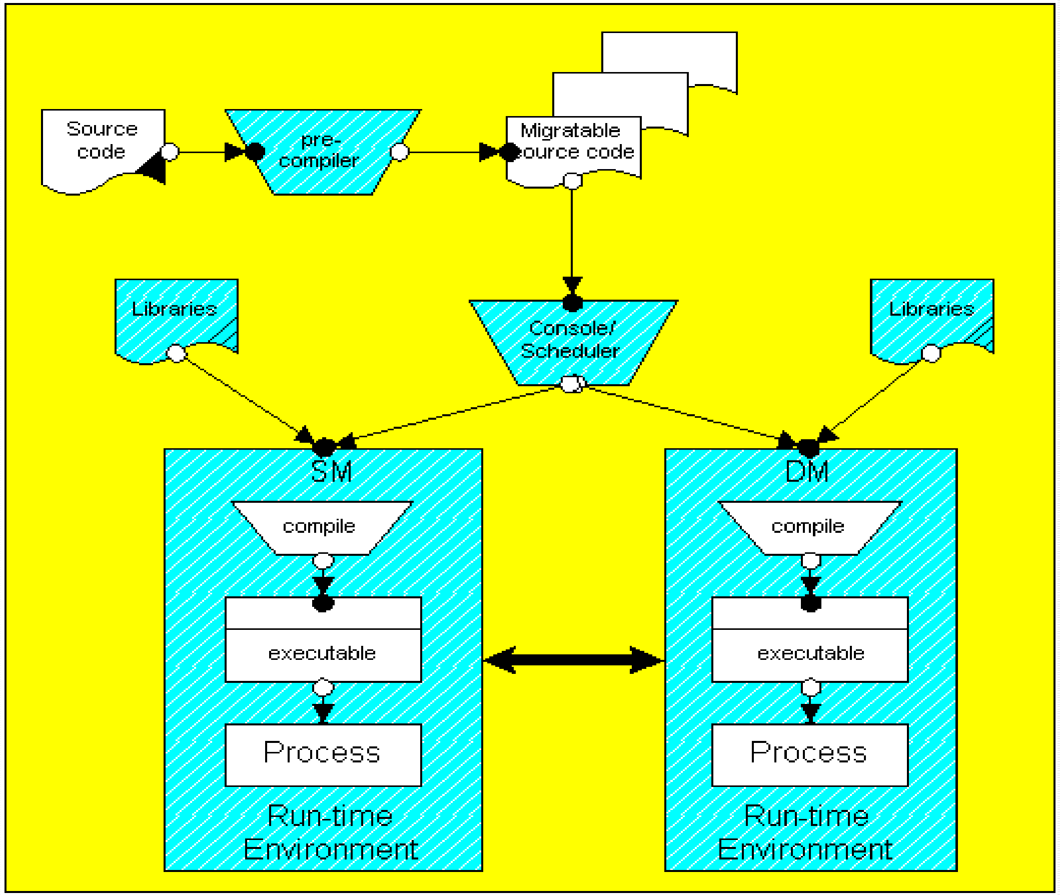
- Internet computing and mobility.
  - Grid, pervasive computing, web services.
  - Traditional programming model.
- Mobility of legacy codes.
  - Legacy codes written in C, C++, Fortran.
  - Heterogeneous computing environment.
- Process migration system helps improve mobility, performance, efficiency and utilization of shared resources.

Process Migration Overview

- Process migration is the act of transferring an active process from one computer to another.
  - Execution state, memory state, communication state.
  - Resources sharing.
- Source machine, destination machine, poll-point, migration point.
- Dynamic preemptive load balancing, fault tolerance, resource access locality

High Performance Mobility Middleware (HPCM)

- Supports user-level heterogeneous process migration.
- Supports mobility of legacy codes.
- A pre-compiler, libraries, a console/scheduler, and a run-time environment.

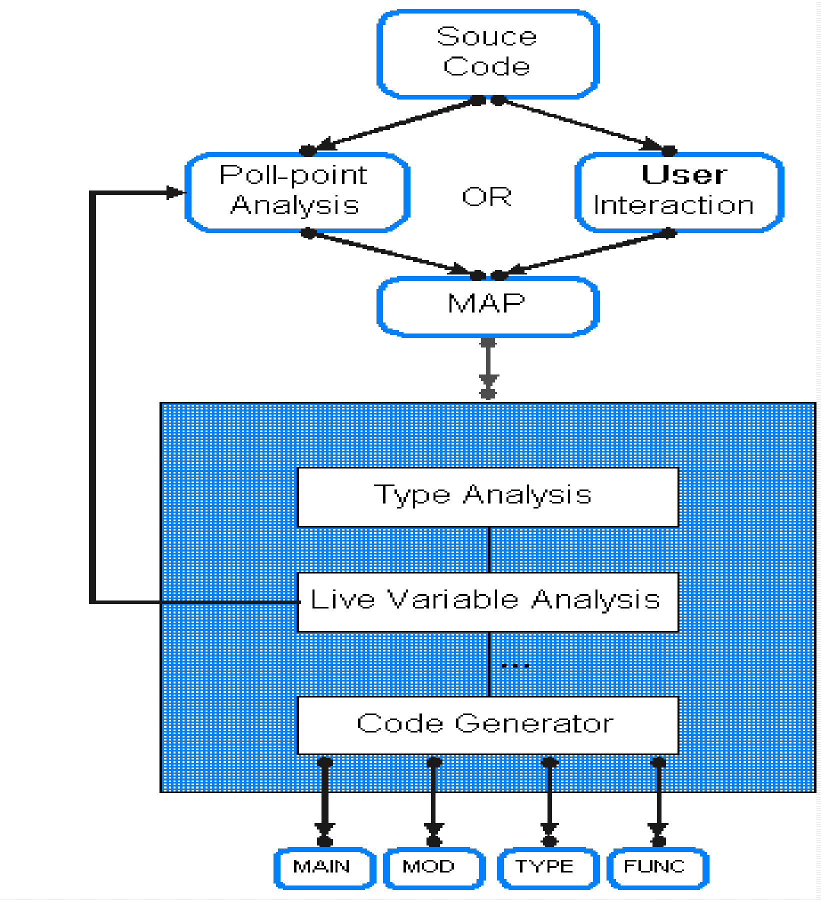


HPCM Components

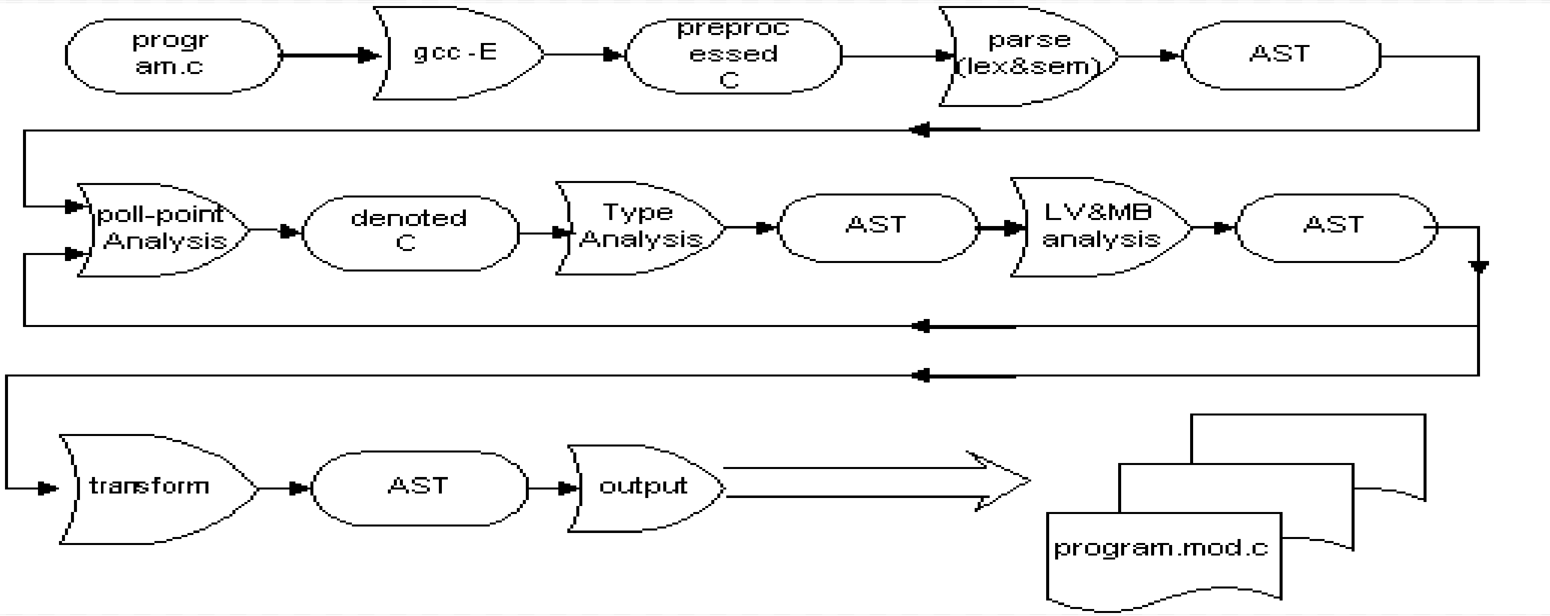
- Pre-compiler
  - Source to source
- Libraries
  - Basic library, communication libraries
  - TCP/IP, PVM, and MPI
- Console/Rescheduler
  - Monitor and coordinate
- Run-time environment
  - deamons

Pre-compiler

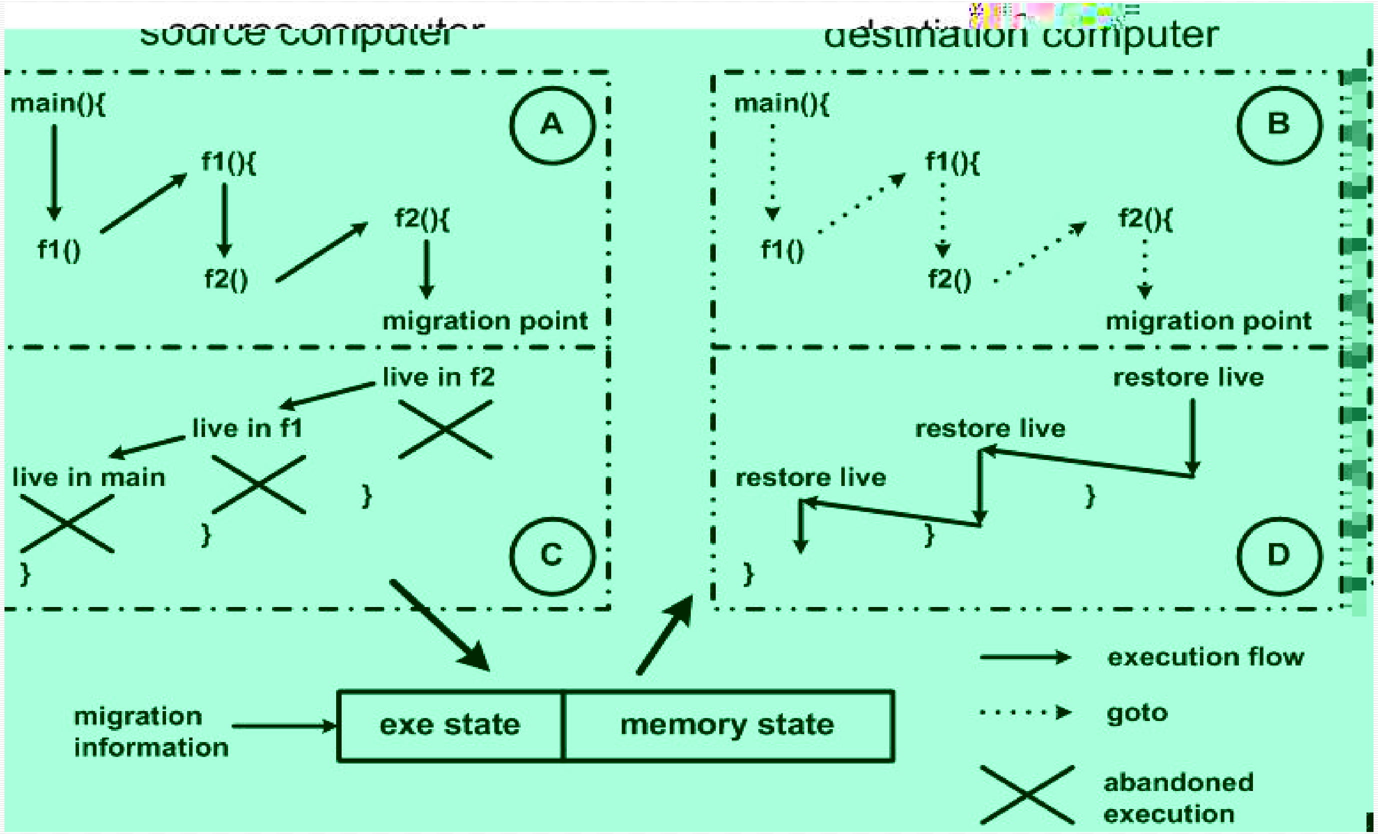
- The pre-compiler is a C-to-C translator, which converts the C source code into its equivalent migration capable C code, and generates related utility files.
  - Execution, memory and communication state transfer.
  - Source Code Annotation



Pre-compiler Workflow



Execution Flow at a Migration Point



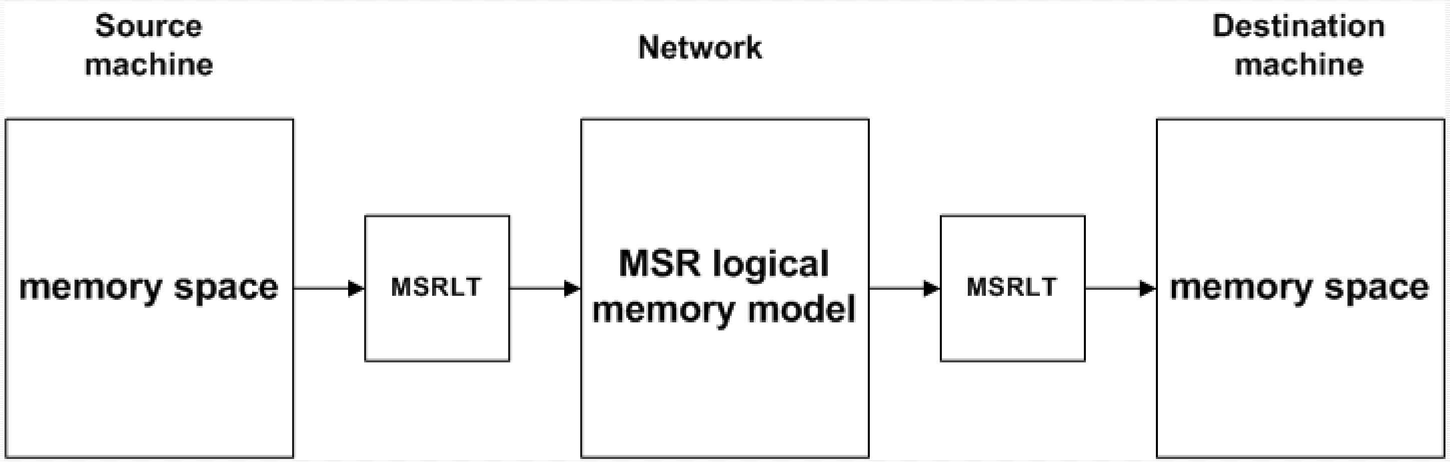
Pre-compiler Functions

- Memory block Analysis and Registration
- Live Variable Analysis
- Type Information Layout Table and Component Layout Table
- Source Code Annotation
- Supplement Files Generation

Memory State Management

- Memory space Registration
  - Global
  - Stack:top of the functions
  - Heap: MSR_MALLOC(), MSR_FREE(), MSR_CALLOC(), MSR_VALLOC(), MSR_REALLOC()
- Data Type
  - Prime
  - Composition: saving & restoring functions
  - pointer
- Memory block Analysis
  - Variables
  - References: registration

Memory Space Representation



- Memory Space Representation Lookup Table
- Type Information Table (TI)
- Component Layout Table

Live Variable Analysis

- Some of those variables will never been used after process migration. Transferring those variables to the destination machine will add unnecessary cost.
- Find out a set of variables whose values are useful in future execution beyond the poll-point.
- To improve the performance of process migration, we try to find out which variable is useful in future execution and which is not. This process is called live variable analysis [4].
- We perform live variable analysis to global variables, local variables and parameters separately to determine the variables that need to be transmitted.

Component Layout Table

```
Struct node {
int x;
double y;
int *t;
struct abb a;
};
```

```
#define Stnode_NUM_COMPONENTS      4
static Component_layout Stnode_
        component_format [ Stnode_NUM_COMPONENTS ] = {
      { offset =0,     TypeInt,      1},
      { offset = sizeof ( int ), TypeDouble,    1},
      { offset = sizeof ( int ) + sizeof ( double ),  TypeIntptr,    1},
      { offset = sizeof ( int ) + sizeof ( double ) + sizeof ( int * ),  Stabb,  1}
}
```

Type Information Table

- Analysis and extract the user-defined type information
- Type Information Table
  - Unique TID
- Recursively lookup the type information

```
/* unit_size,  component_num,  element_num, pointed_type,
  component_format, saving_method, restore_method  */

{ sizeof(struct node),  Stnode_NUM_COMPONENTS,  1,0,
Stnode_component_format, pack_Stnode,unpack_Stnode}
```

An Entry of TI Table

Saving Functions

```
int     pack_Stnode( char * target_mem, int type_id, int length ){
                struct node *refined_target;
                int     i;
                refined_target = (struct node *) target_mem;
                for ( i = 0; i < length; I ++ ) {
                                pack_int( (char *)&(refined_target->x), TypeInt, 1 );
                                pack_double( (char *)&(refined_target->y), TypeDouble, 1 );
                                Save_pointer( (char *) refined_target->t, TypeIntptr);
                                pack_Stabb((char *)&(refined_target->a), Stabb, 1 );
                                refined_target++;
                }
                return 1;
}
```

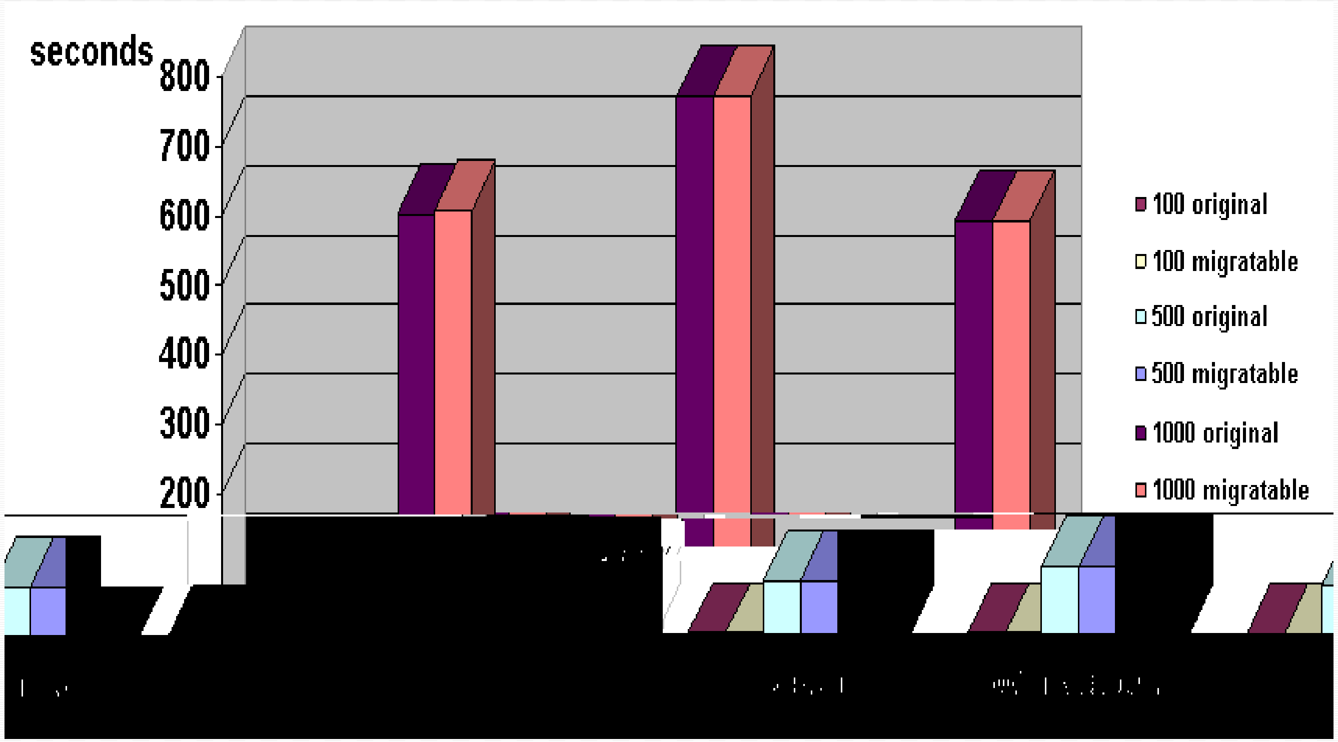
Source Code Annotation

- *head_macro:* puts the functions to the stack of calling sequence, registers memory spaces and memory blocks into MSRLT table.
- *end_macro:* removes functions from the stack of calling sequence.
- *jump_macro:* extracts the calling sequence and jumps to migration point.
- *mig_macro:* For the migrating process, it collects and transmits global variables; for the initialized process, it receives and restores the values of global variables.
- *entry_macro:* It collects and restores local live variables of the current function before entering the migrating point.
- *stk_macro*: It collects and restores the local live variables of the current function after existing the migrating points.

Experiments

- Platforms
  - Sun Blade workstation 100 (W)
    - 1 UltraSparc-IIe 500MHz, 256K, 128MB, SunOS 5.8.
  - Sun Enterprise 450 server (S)
    - 4 UltraSparc II 480Hz, 8M, 4GB, SunOS 5.8
  - Dell Precision Workstation 410MT (L)
    - 2 Pentium III 500MHz, 512K, 768MB, Redhat Linux 8.0
  - Network: 100Mbps internal Ethernet
- Benchmarks
  - Linpack: sequential program translated to C by Bonnie Toy
  - Bitonic by Joe Hummel

Overhead of Process Migration System (Linpack)



Size: 100-1000

Maximum overhead: 0.7% (1000, server)

Homogeneous Process Migration (linpack)

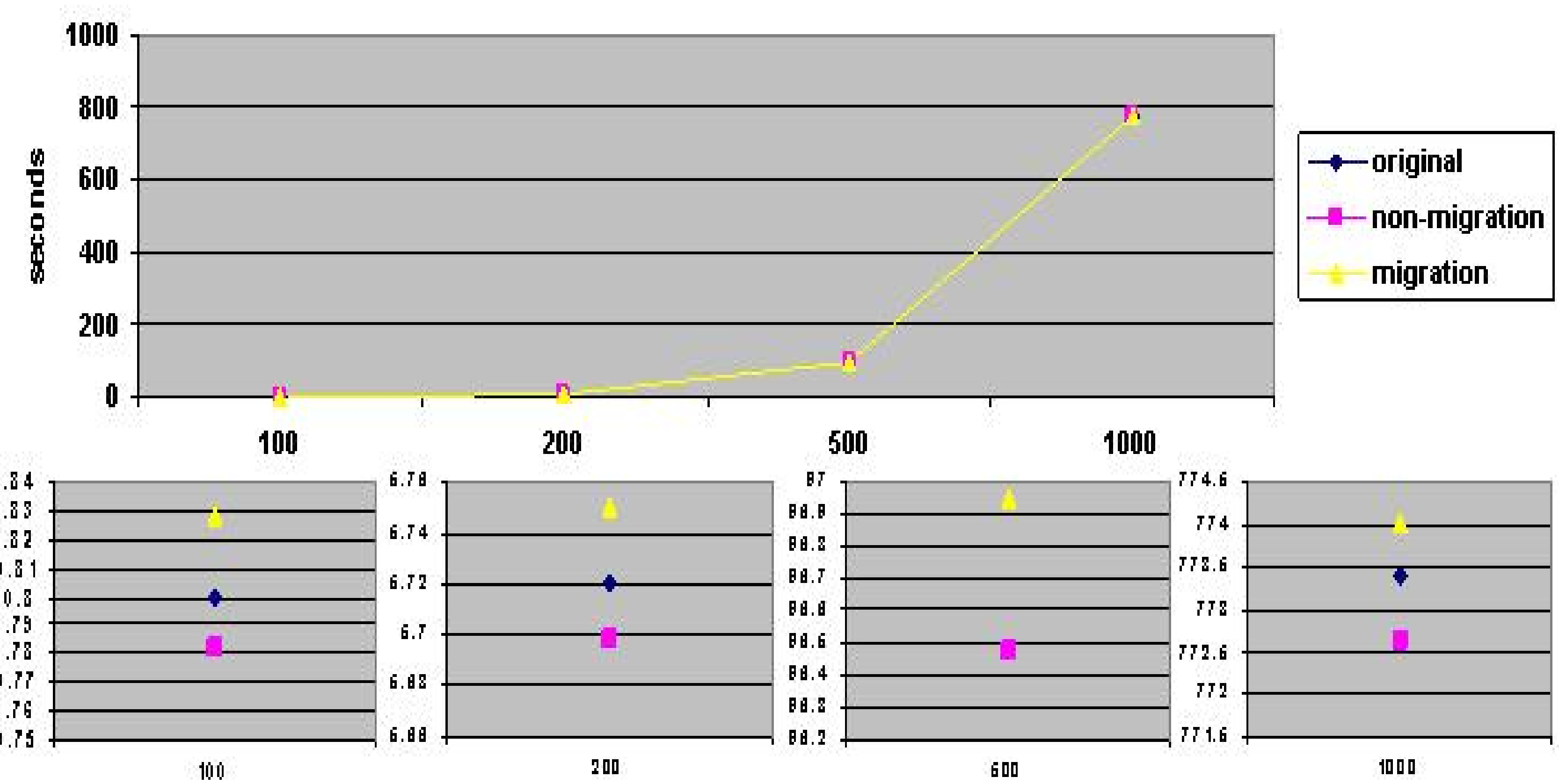
Seconds	100	200	500	1000
original (w)	0.800	5.720	96.475	773.420
original (s)	0.793	5.338	75.628	604.676
non-migration (w)	0.782	5.699	96.478	772.650
non-migration (s)	0.780	5.343	75.353	608.996
migration (w=>s)	0.813	5.464	77.643	622.620
migration (w=>w)	0.828	5.750	96.947	774.026
communication data	82240	323440	2007040	8013040
migration overhead (w)	3.5%	0.5%	0.5%	0.08%

Migration overhead for workstations: 0.08% to 3.5%.

For bigger scales, the overheads are from 0.08% to 0.5%.

For very small application scale, the migration may cause higher overhead.

Migration Overhead of Homogeneous Migration (workstations, linpack)



Heterogeneous Migration From Server to Linux (linpack)

Seconds	500	1000
1. non-migration	75.353	608.996
2. collection	4.708	19.092
3. restoration	4.790	19.334
4. without pipeline 1+2+3	84.851	648.326
5. migration (s=>l)	74.182	610.496

By overlapping the collection, restoration, and transmission, we save 6%-14% of total execution time or almost 50% of the data collection/restoration time.

In this case, migrating a process to a faster machine compensates for the overhead incurred by migration.

Heterogeneous Migration (linux, server, bitonic)

Tree Size	Data size (bytes)			Execution Time (seconds)			Migration Time (seconds)		
level	1	4	8	1	4	8	1	4	8
1024	49416	49932	50620	0.564	0.558	0.570	0.018	0.018	0.018
2048	98568	99084	99772	1.715	1.768	1.786	0.036	0.036	0.047
4096	196872	197388	198076	4.779	4.674	4.742	0.088	0.108	0.108
8192	393480	393996	394684	11.020	11.134	10.994	0.214	0.248	0.253
16384	786696	787212	787900	24.815	24.857	24.724	0.462	0.556	0.557

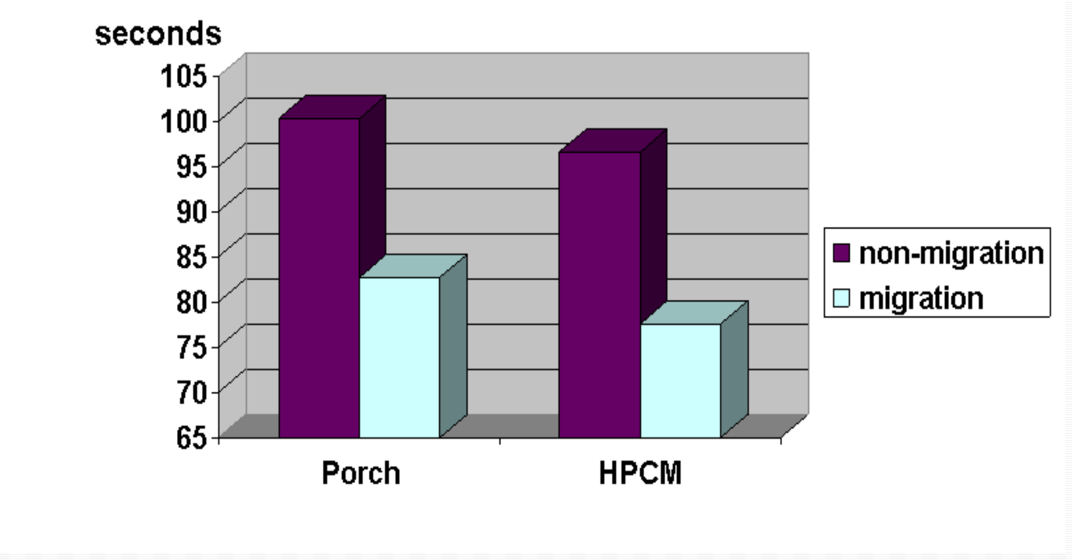
With the increase of the migration point level, the data size increases slowly; the migration time also increases slowly. There is no significant increase for total execution time.

Conclusions and Future Work

- Supporting mobility of legacy codes through heterogeneous process migration.
  - Design of the HPCM middleware and its primary components
  - Design and implementation of the pre-compiler
  - The performance results show that the HPCM middleware is efficient for both the migration and non-migration conditions, and has its real potential in checkpointing as well as in mobility.
- Performance Issues
  - Select migration-point wisely and dynamically
  - Memory State

Questions?

Performance of HPCM and Porch (server to linux, linpack)



- Heterogeneous checkpointing
- Static
- Performance