Process Recovery in Heterogeneous Systems

Kuo-Feng Ssu, *Member*, *IEEE*, W. Kent Fuchs, *Fellow*, *IEEE*, and Hewijin C. Jiau, *Member*, *IEEE*

Abstract—Heterogeneous computing environments, where computers may have different instruction set architectures, data representations, and operating systems, complicate checkpointing and recovery of processes. This paper describes an approach to recovery and an implementation, PREACHES, that provides portable checkpointing of single-process applications in heterogeneous systems using checkpoint propagation. The checkpoint propagation mechanism creates machine-dependent checkpoints for different architectures in the heterogeneous environment. A process is restored on a specific machine with the checkpoint that is appropriate for the architecture. An implementation of PREACHES has been evaluated on a heterogeneous network of workstations, including Sun, HP, and Pentium machines. The experimental results show that PREACHES achieves efficient checkpointing and rapid recovery.

Index Terms—Heterogeneous systems, portable checkpointing, rollback recovery, process migration.

1 Introduction

Numerous checkpointing protocols have been proposed for homogeneous environments, where computers have identical CPUs, operating systems, and data representations [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. Little attention has been devoted to checkpointing and recovery portability.

Traditional system-level checkpointing used in homogeneous systems creates the process state directly from CPUs, heaps, and stacks. The checkpointed process state can be restarted as another process on the same type of the architecture. However, heterogeneous computing systems typically contain incompatible instruction set architectures, registers, data representations, and alignments. Operating systems that manage process information, storage systems, and I/O interfaces are also typically not portable. Therefore, traditional system-level checkpoints of process states cannot be employed on different types of machines.

Application-specific checkpointing has been shown to provide some degree of portability. Programmers may insert checkpointing functions into applications with the assistance of preprocessors or compilers. This approach is not transparent, but it can provide flexibility through enhanced control. For instance, programs may save only the portion of the process state necessary for recovery. Several application-level techniques have been proposed to support checkpointing portability in heterogeneous systems [13], [14], [15], [16], [17], [18], [19]. These techniques extract necessary process states and save them to stable storage in a machine-independent format (Fig. 1a). This approach has two limitations. First, data conversion during checkpointing and recovery potentially increases checkpointing latency, recovery performance, and execution time. Second, due to conflicts with the extraction of specific process state, applications typically cannot utilize page-based checkpointing to reduce overhead [1], [6].

In this paper, *checkpoint propagation*, an approach that enables checkpointing and recovery in heterogeneous systems is described (see Fig. 1b). The method creates machine-dependent checkpoints for each desired architecture. Processes are recovered using the appropriate checkpoint for the target architecture. The mechanism hides data conversion latency and also avoids runtime image regeneration so the recovery process can be efficient. With checkpoint propagation, homogeneous checkpointing tools are able to support heterogeneous checkpointing and recovery. Page-based checkpointing optimization for homogeneous systems, such as copy on write and incremental checkpoints, are also able to enhance the performance of the heterogeneous checkpoint propagation approach [1], [6].

PREACHES, an implementation of checkpoint propagation that achieves recovery of single-process applications in heterogeneous systems, is described in this paper. PREACHES employs an effective primary-backup model for process recovery. PREACHES is able to integrate checkpointing tools, recovery mechanisms, and fault detection schemes. Evaluation of PREACHES was performed with applications written in C and FORTRAN on a network of heterogeneous workstations connected by both ATM and IEEE 802.11 wireless channels. The experimental results show that PREACHES provides low failure-free execution overhead and rapid recovery.

2 Related Work

2.1 Process State Extraction

Extraction of process state is essential for portable checkpointing and heterogeneous recovery. Unlike homogeneous systems, a process state cannot be retrieved directly from memory and CPU. PVM (Parallel Virtual Machine) requires users to define needed critical data for heterogeneous computing [20]. A preprocessor/compiler technique has also been commonly used to place checkpoints and to specify critical data [21], [22], [15], [17], [18]. A preprocessor/compiler examines the source

• K.-F. Ssu and H.C. Jiau are with the Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan 701. E-mail: {ssu, jiauhjc}@ee.ncku.edu.tw.
• W.K. Fuchs is with the College of Engineering, 242 Carpenter Hall, Cornell University, Ithaca, NY 14853-2201. E-mail: wkf3@cornell.edu.

Manuscript received 1 Feb. 2002; revised 7 Sept. 2002; accepted 20 Sept. 2002. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117446.

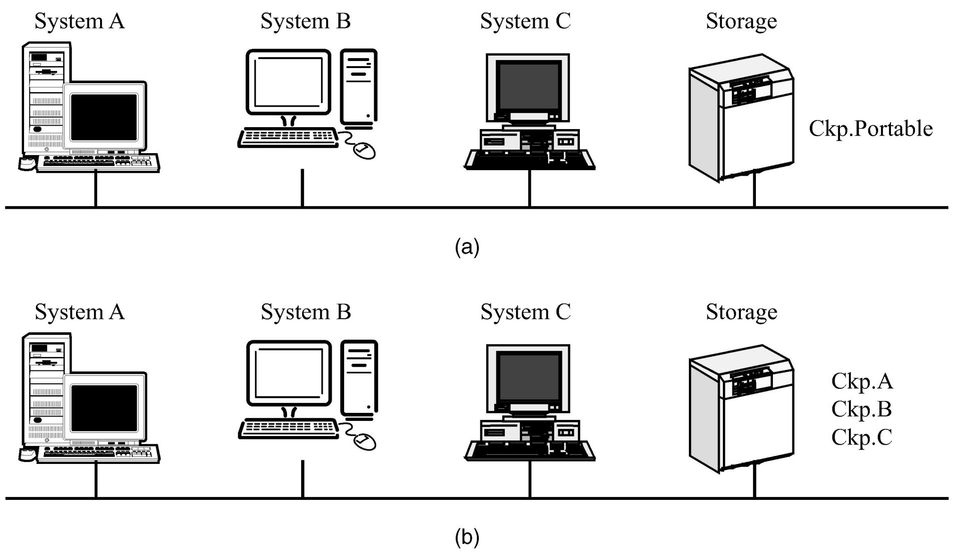


Fig. 1. Two approaches to portable checkpointing. (a) Machine-independent format. (b) Machine-dependent formats.

program and then automatically inserts code into the program for checkpointing and recovery.

2.2 Data Translation

In previous work on portable checkpointing, a process typically stores its state in a machine-independent format. During recovery, the process converts the portable checkpoint into a local data representation based on the target architecture. There are two common machine-independent data formats, XDR (External Data Representation) and UCF (Universal Checkpoint Format). For example, XDR was used in PVM and CosMic [20], [16] and UCF was used in Porch [17]. ASCII data representation has also been utilized [18]. A recompilation technique has also been developed that assumes any process state can be represented by a portable migration program. When a process needs to migrate, its process state is saved in a form of the program. The program is then compiled and executed on the target machine [14].

2.3 Runtime Environment Creation

Some protocols do not include the runtime environment in checkpoints so they need to recreate the correct runtime environment during recovery [17], [22], [15]. A process may record all active procedures during execution. When recovery is initiated, a restarted process calls the procedures in the sequence previously executed to obtain the correct runtime environment. The process of creating an appropriate runtime environment can be simplified if restricted programming models are applied [22], [15]. The program structure of any application must be a loop and checkpointing must be performed at the beginning of the loop in the main procedure. This eliminates the need to save the program counter and stack.

3 Checkpoint Propagation

The checkpoint propagation mechanism of this paper creates a variety of specific machine-dependent checkpoints for the heterogeneous system using a primary-backup process implementation. One alternative would be to replicate complete processes on each type of machine. Each replica could then take checkpoints independently during execution and, thus, the necessary types of checkpoints would be created. This alternative implementation would consume extensive CPU and memory resources, so the performance of the heterogeneous system would be affected. In contrast, our approach exploits a unique primary-backup model for reducing hindrance with other processes.

3.1 System Requirements

Single-process and single-thread applications are assumed. Processes exchange messages through a network. A checkpointing tool is available for each architecture in the system. The fail-stop model is used. A faulty process neither generates erroneous results nor broadcasts them to other processes. Failure types include node crash, system reboot, and unexpected process termination. Floating-point accuracy is not lost due to the conversion of data representations.

3.2 Configuration

In checkpoint propagation, one primary process and at least one simple backup process on a different type of machine are required. The primary process is responsible for all application computation. Backup processes are executed on the specific types of computers that represent the variety of checkpoints desired. For example, assume that we would like to execute an application with checkpointing on a Type A machine but also desire to have checkpoints available for machine Types B and C. In this case, a primary process is executed on the Type A machine and two backup processes are executed on Type B and Type C machines, respectively.

The primary process is responsible for three tasks during failure-free execution: application computation, communication of intermediate results to the backup processes, and checkpointing. The complete computation of an application is performed only by the primary process. When the primary is to be checkpointed, the primary process collects

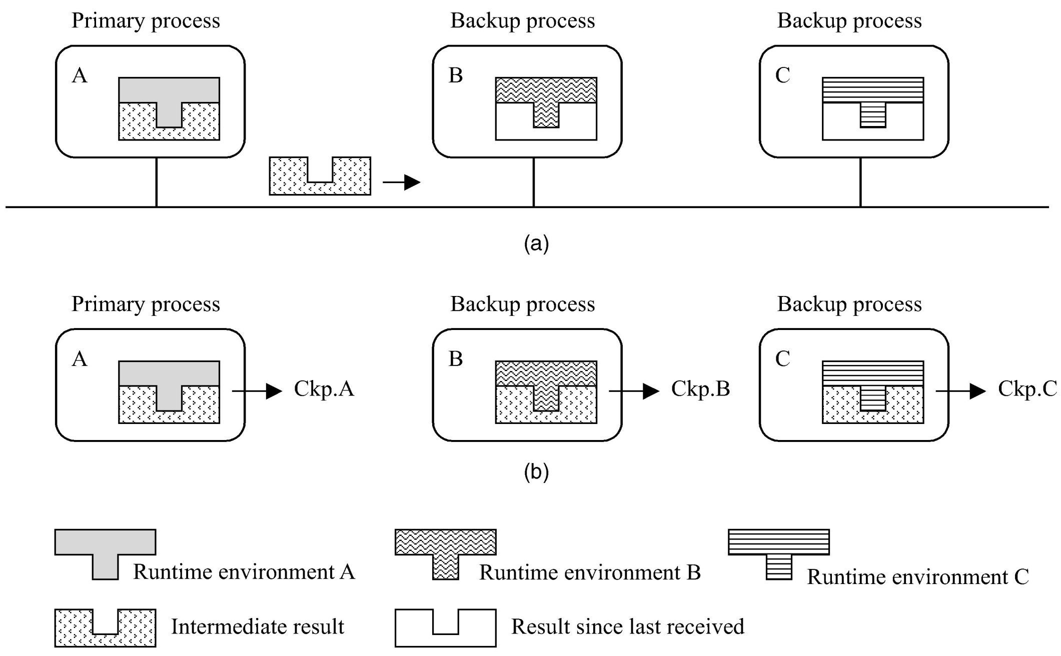


Fig. 2. Checkpoint propagation. (a) Sending intermediate result. (b) Receiving the result and taking checkpoints.

all active variables in the current runtime environment, forwards them to the backup processes, and takes a checkpoint in the local machine format (see Fig. 2a). After checkpointing, the primary process continues execution immediately. The primary process may also monitor the backup processes. If any failure of the backups is detected, the primary process may initiate the recovery mechanism.

The main responsibilities of the backup process are receiving data, generating checkpoints, and monitoring the primary process. The backups do not participate in application computation unless the primary process fails. The backup processes stay idle and wait for the intermediate computation results. The process states of the backups are updated by receiving data from the primary. As illustrated in Fig. 2b, the backups combine their runtime environment and data received to form checkpoints in their local machine formats.

Fig. 3 compares patterns of failure-free execution between the primary and backups. The primary process continuously changes its process state by executing application instructions. The backup processes operate in a passive manner. In the checkpoint propagation scheme, only minor CPU and memory resources are occupied by backup processes since the backups do not explicitly execute applications. The performance of other applications executing on the same machines as backup processes are not significantly degraded, as is illustrated in the evaluation section. Multiple backups will not increase the total amount of messages transmitted in a local network if multicast communication is used. To avoid network contention, the backups save checkpoints in their local disks first. The checkpoints will be flushed to stable storage (remote file server) when network load is low.

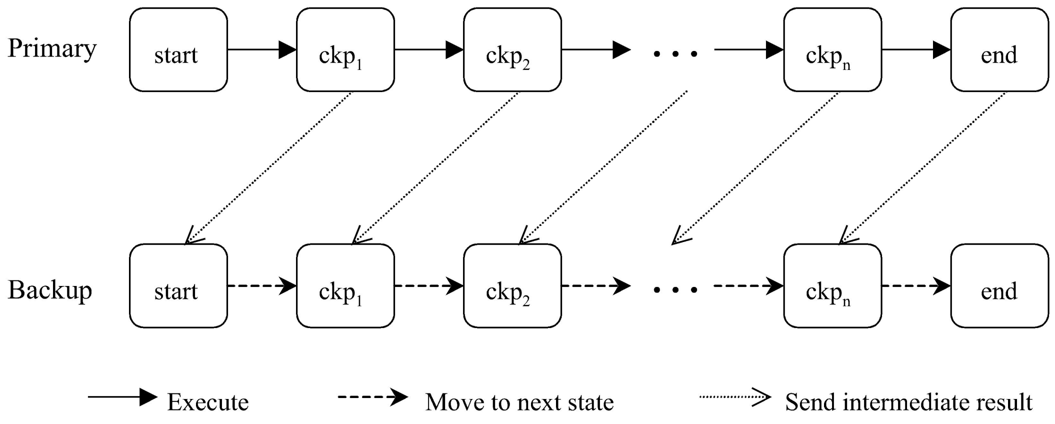


Fig. 3. Execution of primary and backup.

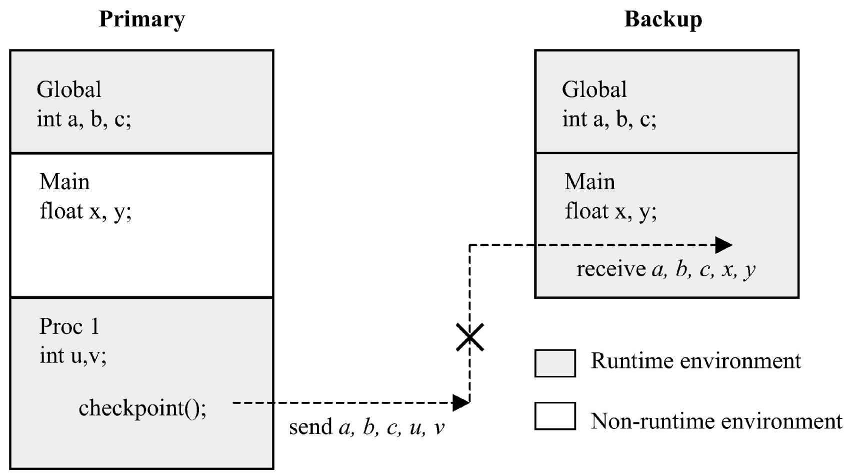


Fig. 4. Inconsistency of runtime environments.

3.3 Consistent Runtime Environment

The runtime environment of a process typically changes during the application execution. As described earlier, the backup processes implementing checkpoint propagation integrate intermediate results received with their runtime environments to create checkpoints. Therefore, it is important that the backup processes maintain runtime environments that are consistent with the primary process during checkpointing. For example, in Fig. 4, assume that a primary process enters Procedure 1 and a backup process remains in Procedure Main. The primary process sends computation results to the backup process. However, the backup cannot integrate the intermediate computation results correctly due to the different runtime environments. A simple solution is for the backup process to track the primary whenever the primary changes its runtime environment.

A message of `enter procedure` or `exit` is used to notify the backup processes when the primary changes its runtime environment. For the notification of `enter procedure`, the primary transmits intermediate results and a target procedure to the backups. The backup processes then update their data and enter the target procedure. For the notification of `exit`, the backups follow the primary process to exit the current procedure.

3.4 Reducing Unnecessary Notifications

Some notifications for updating runtime environments in the backup processes during checkpointing are not necessary. Consider an application with a recursive procedure that does not contain any checkpointing function. In this case, the runtime environment with recursion is not needed for checkpointing. Therefore, the primary process does not have to notify the backups when the primary enters the recursive procedure.

Necessary procedures are defined as the procedures that have checkpointing instructions or static variables. *Sufficient procedures* are the procedures in a chain of active procedure calls when a process enters any necessary procedure. From the definitions, if a primary process enters a sufficient procedure, checkpointing or updating values of static variables may possibly be performed in the future. Whenever the primary enters or exits a sufficient procedure, the backup processes must be notified.

Sufficient procedures can be easily obtained. A directed graph G, that represents the relationship between procedure calls, can be constructed using the following rules:

1. Each procedure represents a vertex in the directed graph G.
2. A directed edge (a, b) exists from vertex a to vertex b in graph G if procedure b is called by procedure a in the application.

In the example of Fig. 5, vertex A represents a main procedure and vertex K is a recursive procedure. Let V be a set of vertices in G and V' be a subset of V that represents necessary procedures. A vertex that leads to any vertex in V' means that the procedure it represents is a sufficient procedure. In the example, the sufficient procedures are A, B, C, D, E, F, and H. The backup processes will be notified only when the primary enters the sufficient procedures.

An alternative to the above approach is to have the primary process request that backups update their runtime image completely instead of incrementally. To implement this, the primary would manipulate a shadow stack and record the calling path at the time of execution. The primary would transmit both the shadow stack and the active procedure call sequence to the backup processes, so the backups can reach a consistent runtime environment. The main drawback of the technique is that an application requires at least twice as much memory due to the shadow stack.

3.5 Message Logging

Section 3.2 required that each recovering architecture run a backup from the beginning of the computation. However, machines with different architectures may become available during an execution. Message logging can enable machines to join the backups [23], [24]. Stable storage logs each message sent by the primary process. When a process from a different type of machine needs to join the backups, it can replay the messages stored in the stable storage and also take the last checkpoint. The process will then have the same state as other backups.

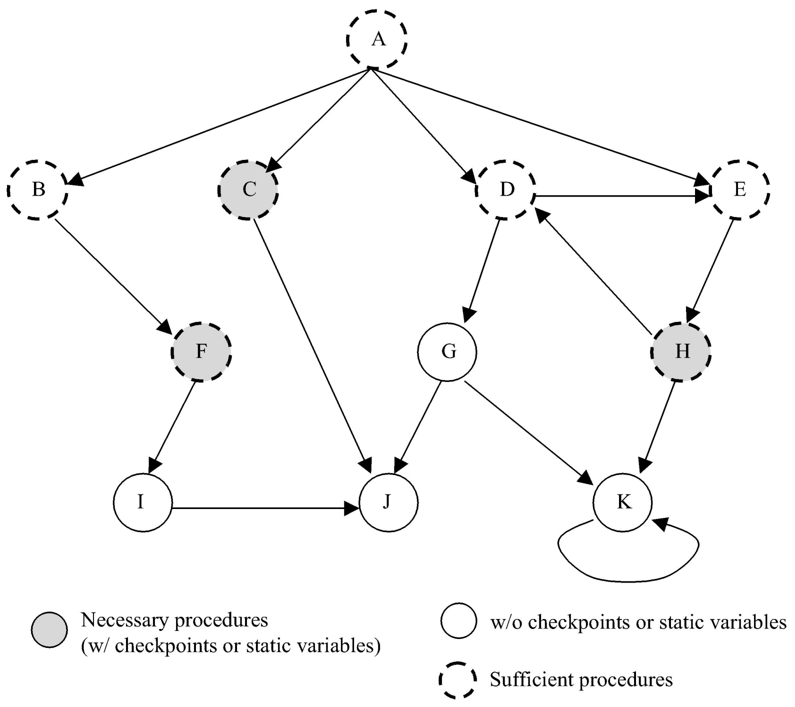

A
B
C
D
E
F
G
H
I
J
K
Necessary procedures (w/ checkpoints or static variables)
w/o checkpoints or static variables
Sufficient procedures

Fig. 5. Relationship between procedures.

3.6 Process Migration

Checkpoint propagation supports not only process recovery but also process migration in heterogeneous environments. Our mechanism requires that possible migration points be inserted in user applications [13]. With appropriate arrangement of a primary and backups, processes can be migrated among different architectures. A detailed algorithm for process migration is presented in Fig. 6.

4 PREACHES

PREACHES is an implementation of checkpoint propagation designed for C and Fortran programs with Unix dialects. There are mature checkpointing tools for Unix, such as Condor [2], Libckpt [6], and RENEW [25], which we are able to utilize.

4.1 Data Conversion

Data conversion is an important component in the communication between primary and backup processes. For example, there can be different byte orderings within a word. Sun processors use *big endian* byte order and Intel processors use *little endian*. Most previous work has relied on converting data to a portable format at the sender and translating data to a local format at the receiver. PREACHES utilizes a receiver-based mechanism to eliminate the overhead of data conversion at the sender (primary process). In our approach, the primary process sends messages without immediate translation. The backup processes identify the primary process and the architecture used after data arrival. The backup processes convert the received data to their local representations. With this technique, the primary process resumes its execution immediately after sending intermediate results to the backup processes (nonblocking). The overhead for data conversion by the primary process is eliminated. Receiver-based data conversion has the disadvantage that each backup process must record all the data formats for every architecture.

In PREACHES, data conversion is implemented as a library. Several basic data types, including `char`, `int`, `long`, `float`, `double`, etc., were developed for PCs, Sun, and HP workstations. Programmers can use the primitive types to further support the data type `struct` and `union`.

4.2 Pointer Translation

Pointer translation is implemented using *address space tables* in PREACHES. The address space table records the beginning address, size, and number of elements for each variable declared in the source program. Fig. 7 shows the structure of the address space tables. There are four attributes for each variable in the table:

- *id*: A unique identification number for a variable.
- *address*: A starting virtual address of a variable. The address is obtained by the instruction, `address=&variable`.
- *size*: The size of the variable. The size is computed by the `size=sizeof(variable)` instruction.
- *element*: The number of elements in the variable. The number is defined in the source code.

When a primary process starts execution or invokes a procedure call, the primary sends address space information to the backup processes. The backups modify the address information of the primary based on the data received. Each backup process also manages its own address space table. The backups access the two address

```
Definition:
    $N = \{W_1, W_2, ...\}$, set of different machine types
    $\mathcal{P}(N)$: the power set of $N$
    $M$: set of possible target machine types, $M \in \mathcal{P}(N)$
    $W_i$: set of type i machines, $W_i \cap W_j = \emptyset, i \neq j$
    $B$: set of machines performing as backups, $|B| = |M| - 1$
    $p_i$: type i machine executing a primary process, $p_i \in W_i$
    $b_i$: type i machine with a backup, $b_i \in W_i \cap S$
    $t_i$: type i target machine, $t_i \in W_i - \{p_i\}$
Initialization:
    Choose $U = W_i \in M$ and pick a $p_i \in U$ to execute a primary
    $\forall W_i \in M - \{U\}$, pick a $b_i \in W_i$ to serve as backup
Migration:
    (i) $p_i \rightarrow t_i, p_i \neq t_i$:
        $p_i$ takes a checkpoint
        $p_i$ terminates
        $t_i$ restarts using the checkpoint taken by $p_i$
        $t_i$ promotes itself to $p_i$
    (ii) $p_i \rightarrow t_j, i \neq j$:
        $p_i$ sends intermediate data to $b_j$
        $p_i$ is demoted to a backup process $b_i$
        if $(b_j = t_j)$ $b_j$ promotes itself to $p_j$
        else {
            $b_j$ takes a checkpoint
            $b_j$ terminates
            $t_j$ restarts using the checkpoint saved by $b_j$
            $t_j$ is promoted to $p_j$
        }
```

Fig. 6. Algorithm for process migration.

space tables and translate pointers to the corresponding local addresses.

In Fig. 7, in addition to its own address space table (Table B), the backup maintains the address space table from the primary process (Table A). Table A is sorted by attribute address. Assume that a pointer ptr points to address 2036 in the primary and needs to be transferred to the backup process. By comparing the address of each entry in Table A with 2036, it is determined that ptr points to one of the elements in id 2 which represents variable b. The offset is computed by: offset=(ptr-address[id]) /size[id]. The offset of variable b is thus (2036-2004)/4, which is 8. Therefore, ptr points to b[8]. According to the Table B, b[8] can be further translated to the local address of the backup process by the id number and the offset: local_address=address[id]+ offset*size[id]. The value of ptr in the backup is (5608+8*4), which is 5640.

4.3 Dynamic Memory Allocation

PREACHES supports dynamic memory allocation in user applications. A primary process can allocate and release memory for the application during execution. Intermediate computation sent by the primary process may contain dynamically allocated memory. Backup processes have no knowledge of the allocated memory since they do not directly participate in the computation. Therefore, details of the memory allocation must be provided by the primary process. The primary has to send the data type of the allocated memory, the number of data elements in the memory, and the content of the memory to the backup processes. According to the transmitted information, the backups allocate memory, update address space tables, translate the received memory content into local data representations, and store them in the allocated memory. Checkpointing or updating the runtime environment can then be resumed. The primary process notifies the backups to release memory if the allocated memory is released by the primary.

4.4 Checkpointing

Libraries and tools developed for homogeneous checkpointing can be directly used in PREACHES. The primary and backup processes take checkpoints individually in their specific machine formats, so the checkpointing does not involve issues beyond the local machine's environment. The current implementation of PREACHES uses RENEW [25] and libckpt [6] checkpoint libraries.

Source program

```
int a;
float b[10];
double x[20];
double y;
long z[4];
char str[100];
```

Table A (Primary space)

id	address	size	element
1	2000	4	1
2	2004	4	10
3	2044	8	20
4	2204	8	1
5	2212	4	4
6	2228	1	100

Table B (Backup space)

id	address	size	element
1	5604	4	1
2	5608	4	10
3	5648	8	20
4	5808	8	1
5	5816	8	4
6	5848	1	100

Fig. 7. Two address space tables in a backup process.

```
Table Maintenance                     Data Transmission & Conversion

void push_primary(x,y,z);             int preaches_pkint(val_addr, len);
void del_primary(item);               int preaches_upkint(val_addr, len);
void sort_primary(front,rear);        int preaches_pkuint(val_addr, len);
void proc_primary();                  int preaches_upkuint(val_addr, len);
void push_backup(x,y,z);              int preaches_pklong(val_addr, len);
void del_backup(item);                int preaches_upklong(val_addr, len);
void push_counter(x);                 int preaches_pkbyte(val_addr, len);
void del_counter();                   int preaches_upkbyte(val_addr, len);
void go_up();                         int preaches_pkfloat(val_addr, len);
void print_primary();                 int preaches_upkfloat(val_addr, len);
void print_backup();                  int preaches_pkdouble(val_addr, len);
void print_counter();                 int preaches_upkdouble(val_addr, len);
void recv_addrtbl();
                                                 Process Connection
Pointer Translation
                                      void primary_setup();  void proc_end();
unsigned translate(addrs);            void backup_setup();   void main_end();
```

Fig. 8. Examples of PREACHES functions.

User-triggered checkpointing is utilized in PREACHES. Programmers, preprocessors, or compilers directly insert checkpointing instructions, (e.g., `checkpoint_here()`) into the desired locations of the source code. Besides, a checkpoint instruction (forced checkpoint) must be placed for each output commit. The user-triggered mechanism ensures that the primary and backups know exactly when and where to take checkpoints. Therefore, the primary and backups can maintain consistent checkpoints even though code optimization is applied during compilation.

4.5 Recovery

The PREACHES implementation provides two methods of recovery for the primary process. The process is recovered by the latest checkpoint or an active backup process. In the first alternative, the restarted process reads the checkpoint file from the disk, reconstructs its heap and stack, and invokes a `longjmp()` system call. Recovery time is dominated by the time necessary to read the checkpoint from the disk, which is typically proportional to checkpoint size.

In the second approach to recovery, PREACHES promotes one of the backup processes to a primary process according to a predefined priority. This requires few instructions, so execution can be restarted almost instantaneously. In general, recovery through a backup process as described above is considerably faster than reading checkpoint files from stable storage.

Failures in backup processes are recovered either by restarting the backup from its checkpoint or by simply removing the process, if sufficient recovery backup processes or checkpoints are available. The PREACHES implementation uses TCP/IP communication between primary and backup processes. Hints from the socket layer are utilized to detect restricted classes of process and communication failures [26].

4.6 Code Instrumentation

Like other application-level checkpointing schemes, our technique is nontransparent. Functions provided by PREACHES are placed in source code to incorporate into applications (see Fig. 8). In the beginning of the main program, `process connection functions` are inserted to establish network connection between a primary and backups. Application programmers should decide where to take checkpoints and then place checkpointing functions in correspondent locations. `Data transmission and conversion functions` are added in the locations of checkpoints and sufficient procedures. Therefore, the primary can send the value of each variable to the backups. The backups make data conversion if their representations do not match the primary's. If applications contain pointers, `table maintenance functions` are needed in both main procedure and sufficient procedures. `Pointer translation functions` should be included at each checkpoint and sufficient procedure. Embedding PREACHES functions into an application requires careful programming and complicates debugging. Previous work exploited preprocessors to place needed code automatically [21], [22], [15], [17], [18]. Preprocessors not only reduce the work of programmers, but also improve the accuracy of code instrumentation.

4.7 Limitations

PREACHES demonstrated an implementation of the checkpoint propagation targeted for C and Fortran languages. The current version does not support file I/O or multiprocess applications. For other programming languages, PREACHES functions and correspondent preprocessors need modifications. For example, public methods with object serialization are required for passing objects in Java. However, some code instrumentation may violate principles of object-oriented programming such as encapsulation.

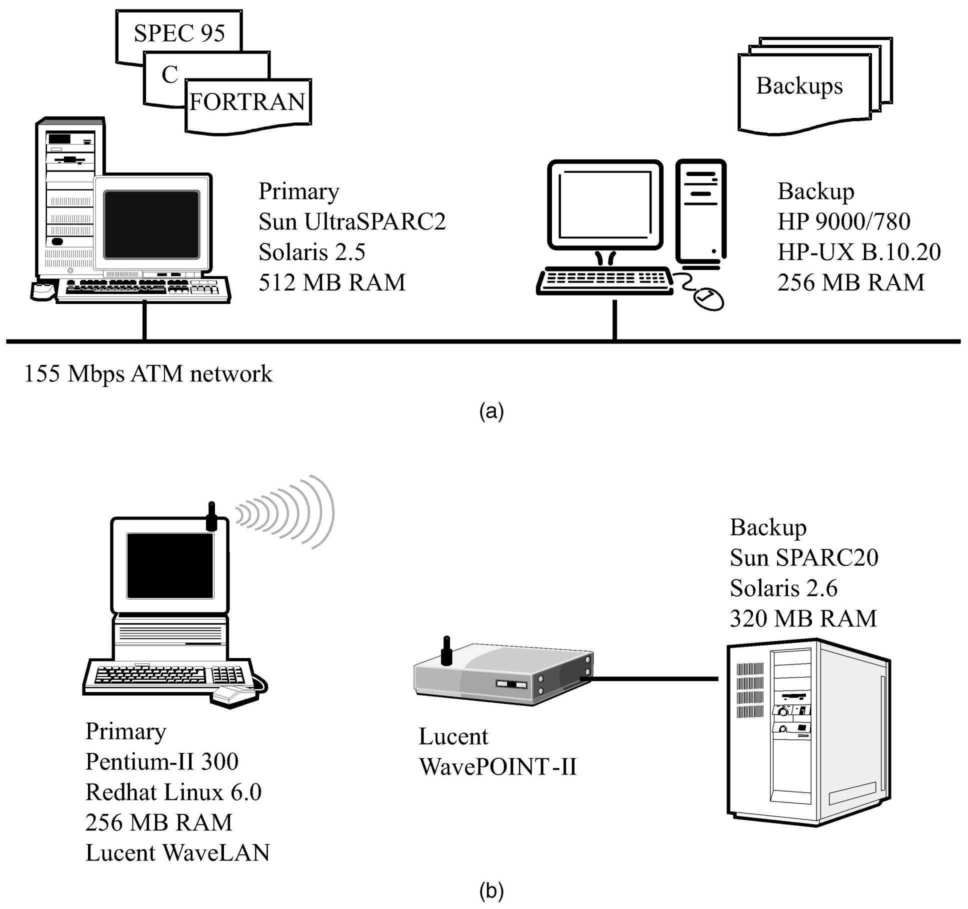


Fig. 9. Experiment environments. (a) Wired network environment. (b) Wireless network environment.

5 Evaluation Context

5.1 Environment

Performance was measured in two heterogeneous environments, one with a high performance wired network and the other with a relatively low bandwidth wireless network. The first set of experiments were performed on a Sun UltraSPARC 2 with Solaris 2.5 and an HP 9000/780 with HP-UX B.10.20 (see Fig. 9a). The Sun had 512 MB of memory and the HP had 256 MB memory. The workstations were connected by a 155Mbps ATM network. For each application, one primary process executed on the Sun UltraSPARC 2 and a backup process was on the HP 9000/780.

The second set of experiments were implemented on a heterogeneous wireless network, as illustrated in Fig. 9b. A Sun SPARC20 with Solaris 2.6 and a Pentium-II 300 MHz PC with Redhat Linux 6.0 were connected by IEEE 802.11 wireless devices, including a WavePOINT-II access point and 2Mbps WaveLAN interfaces. The Sun workstation had 320 MB in memory and the PC had 256 MB in memory. The Pentium PC executed the primary processes and the backup processes were on the Sun SPARC20 workstation.

5.2 Applications

The performance of PREACHES was evaluated using seven CPU-intensive applications. `tomcatv`, `swim`, and `mgrid` are SPEC95 benchmarks [27]. `flops20`, `nsieve`, and `tfftdp` are from BenchWeb [28]. `btree` was developed by the authors for evaluating applications with a large number of pointers. `mgrid` was implemented in FORTRAN and the other applications were written in C.

- *tomcatv* is a two-dimensional vectorized mesh generation program. It has a variety of memory access patterns and performs little I/O.
- *swim* is a solver for shallow water equations using single precision finite difference approximations on a 512*512 grid. It contains several memory access patterns and a high cache hit rate.
- *mgrid* is a simplified multigrid solver for a three-dimensional potential field. It solves constant coefficient equations.
- *btree* constructs a 5-million-node binary tree. This application, containing 10 million pointers, evaluates pointer translation between heterogeneous machines.
- *flops20* evaluates the performance of processing floating-point numbers. The program estimates the MFLOPS (million floating-point operations per second) for FADD, FSUB, FMUL, and FDIV operations based on an instruction mix scheme.
- *nsieve* is a prime-number generator. This program extensively uses long integers and varying size arrays.
- *tfftdp* computes the Fast Fourier Transform in double precision.

TABLE 1
Execution Time and Checkpointing Overhead with the Wired Network

Program	Unmodified	PREACHES w/o Checkpoint		PREACHES w/ Checkpoint		Primary/Backup
	Exec Time (sec)	Exec Time (sec)	Ovh (%)	Exec Time (sec) (# of Ckp)	Ovh (%)	Ckp Size (KB)
btree	910.5	912.5	0.2%	942.3 (3)	3.5%	58652/58653
flops20	640.3	641.8	0.2%	644.7 (2)	0.7%	60/52
nsieve	523.0	523.7	0.1%	531.2 (2)	1.6%	25030/52
swim	865.5	868.2	0.3%	881.1 (3)	1.8%	14505/14500
tfftdp	1027.3	1028.8	0.1%	1056.5 (3)	2.8%	65588/65595
tomcatv	904.3	906.0	0.2%	935.0 (3)	3.2%	2157/2135
mgrid	1294.6	1296.0	0.1%	1299.1 (4)	0.4%	7617/7517

Functions provided by PREACHES were inserted in the applications for the experiments. The added functions were for establishing connections, maintaining address tables, specifying intermediate computation results, translating data, and ensuring backup processes had the correct execution path. At the time of the experiments, PREACHES was not implemented with a preprocessor, so required additional code was inserted manually. In the experiments, 35 percent additional lines were inserted in the original source code. This increase in code size was due to the large number of variables used in the benchmarks.

6 Experimental Results

6.1 Checkpointing Overhead

In our experiments, the checkpoint interval was set to approximately 5 minutes. Table 1 summarizes the execution overhead of the primary processes with PREACHES on the wired network environment. When the primary processes transmitted intermediate computation results to backup processes, but did not take a local checkpoint, the total execution time was delayed less than 0.3 percent. When the primary process both took local checkpoints and interacted with the backup processes, the average performance overhead of the benchmarks was 2 percent and the worst case overhead was 3.5 percent.

Checkpoints from the primary and backup processes can be very dissimilar in size. In `nsieve`, the checkpoint of the primary was approximately 500 times larger than that of the backup. This was a consequence of the primary dynamically allocating more memory than the backup process. In the dynamic memory allocation of the application, the heap size was increased by `malloc` calls, but not released by `free`. In the original application, memory exclusion could have been applied to reduce the checkpoint size.

Message traffic is a factor in the overhead. The size of messages for the applications ranged from 0.5 KB to 65 MB, as illustrated in Table 2. Checkpoints were inserted only in the main procedure for most applications, so the number of messages was significantly reduced. In `btree`, all unnecessary messages in the recursive procedures were eliminated.

In some nonintuitive cases, the performance was not primarily due to checkpointing and messages sent to backup processes. For example, in Table 1, `tomcatv` had significantly smaller message size and checkpoint size than `tfftdp`, but `tomcatv` had a larger overhead. From an inspection of the assembly code generated for `tomcatv`, it was discovered that the compiler code optimization was highly sensitive to the extra instructions inserted in the `tomcatv` source program.

Table 3 shows the checkpointing overhead using PREACHES in the wireless environment. The performance was degraded due to intermediate results sent from the primary to backup processes. Transferring checkpoints through the wireless channel introduced considerable performance overhead. For the application `btree`, the primary process spent 795 seconds to send four 35 MB checkpoints and the execution overhead grew to 62 percent.

TABLE 2
Size and Number of Messages

Program	btree	flops20	nsieve	swim	tfftdp	tomcatv	mgrid
Mesg Size (KB)	58594	0.8	0.5	14448	65536	2083	7101
Reduced (# of mesg)	4	5	3	4	4	4	5
Original (# of mesg)	180000040	11	35	724	175	4	115253

TABLE 3
Checkpointing Overhead in the Wireless Environment

Program	Unmodified	PREACHES w/ Checkpoint		Checkpoint Size
	Exec Time (sec)	Exec Time (sec) (# of Ckp)	Ovh (%)	(KB)
btree	1227.4	2073.3 (4)	62.31%	35212
nsieve	901.8	902.8 (3)	0.12%	56
swim	1245.8	1578.9 (4)	26.75%	14505
tfftdp	1052.3	1100.9 (3)	4.62%	4152
tomcatv	1054.4	1229.4 (3)	16.60%	7450

Nonblocking message transmission was implemented to improve performance. With the nonblocking scheme, the checkpointing overhead was significantly reduced. The total execution time of `btree` was reduced approximately 700 seconds (33 percent). Fig. 10 shows the performance improvement using nonblocking message transmission.

Due to the similarity of our results for the wired and wireless environments in Sections 6.2, 6.3, and 6.4, the remainder of the paper shows only the experimental results collected in the wired environment.

6.2 Overhead Comparison with Libckpt

The performance of the checkpointing component of PREACHES is competitive with checkpointing tools developed for homogeneous computing. Fig. 11 compares the overhead between libckpt for homogeneous systems and PREACHES for heterogeneous recovery. Libckpt generated checkpoints for Sun Solaris and PREACHES created checkpoints for both Sun Solaris and HP-UX. PREACHES required twice as much disk space for storing the heterogeneous checkpoints compared to libckpt. In our experiments, copy-on-write was utilized to reduce the checkpointing overhead for both PREACHES and libckpt. The results show that the average checkpointing overhead of libckpt was 1.50 percent and the average overhead of PREACHES was 2.07 percent.

6.3 Backup Process Overhead

The impact of backup processes executing on the same machine as other applications was also examined. Three SPEC benchmarks (`mgrid`, `swim`, and `tomcatv`) were run on the HP machine with 256 MB main memory. At the same time, several backup processes for `swim` were also executed on the HP workstation. The increased total execution time of the SPEC benchmarks shown in Table 4 indicates the performance lost due to the backup processes. The average execution overhead for one backup process was 0.24 percent and the average overhead for six backup processes was

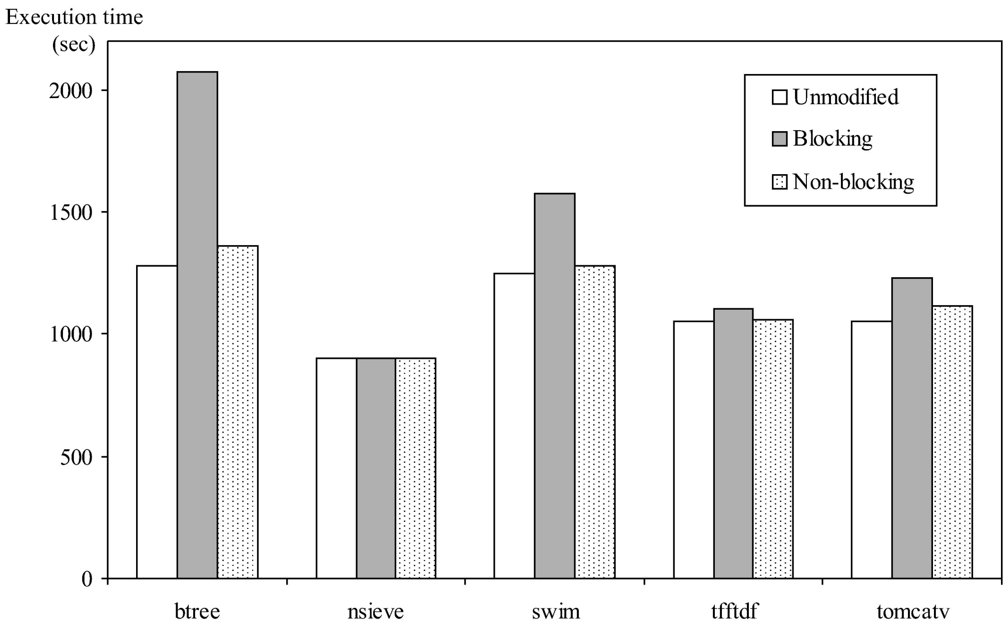


Fig. 10. Performance improvement.

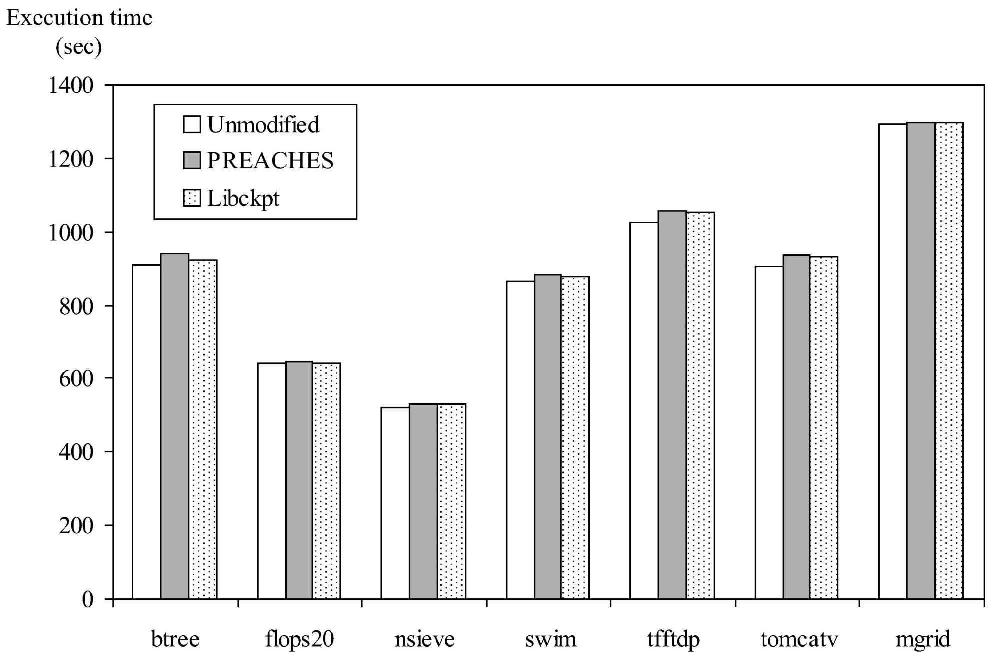


Fig. 11. Comparison between PREACHES and Libckpt.

2.81 percent. The results show that checkpoint propagation in PREACHES is considerably more efficient than simple process replication.

The overhead due to backup processes is caused by two dominant factors. First, checkpointing to disk can interfere with the execution of other applications running on the same machine. Second, a backup process can compete with other applications for limited memory resources.

TABLE 4
Overhead Caused by Backup Processes of `swim`

Program	Execution Time (sec)						
	Number of swim backup processes						
	0	1	2	3	4	5	6
mgrid	804.9	806.8	809.2	812.7	817.8	821.1	827.3
swim	675.0	677.0	680.1	683.6	687.3	690.2	697.1
tomcatv	529.4	530.3	531.4	533.3	536.1	538.1	541.5

TABLE 5
Restart Time

Program	Local Ckp (sec)	Remote Ckp (sec)	Backup Process (sec)	Ckp Size (KB)
btree	1.54	5.67	3.87e-4	58653
flops20	0.01	0.02	3.85e-4	52
nsieve	0.01	0.02	3.83e-4	52
swim	0.38	1.40	3.86e-4	14500
tfftdp	1.68	6.03	3.85e-4	65595
tomcatv	0.07	0.26	3.83e-4	2135
mgrid	0.22	0.86	4.34e-4	7517

6.4 Restart Time

Restart time is the time from when process recovery is first initiated to the point when the process has fully resumed execution from its latest checkpoint or backup process. Three implementations of heterogeneous recovery were measured. Applications were executed on the Sun workstation and process recovery was initiated on the HP workstation by local checkpoints, remote checkpoints, and backup processes. A comparison of the results is shown in Table 5.

The restart time, based on backup processes, dramatically outperformed the other two implementations. An exception handler only needed to promote the backup to become the primary. All the benchmark applications were able to implement recovery within 1 millisecond using backup processes.

7 Summary

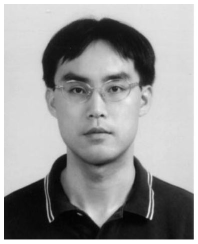
This paper demonstrates that the checkpoint propagation mechanism can support heterogeneous recovery and migration using homogeneous checkpointing libraries. Dynamic memory allocation and pointers can be utilized in the user applications and page-based checkpointing optimization is allowed to improve performance. PREACHES, a heterogeneous checkpointing and recovery tool, was implemented and evaluated. A receiver-based mechanism was developed to reduce the overhead of data conversion and pointer translation. Benchmark code with PREACHES was evaluated on both a wired and a wireless network. The experimental results show that PREACHES achieves low checkpointing overhead and fast recovery in heterogeneous systems.

Acknowledgments

This research was supported in part by the US Defense Advanced Research Projects Agency (DARPA) under contract DABT 63-96-C-0069, in part by the US Office of Naval Research under contract N00014-97-1-1013, and in part by the National Science Council under contract NSC 89-2218-E-006-151.

References

[1] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, "The Performance of Consistent Checkpointing," *Proc. IEEE Reliable Distributed Systems Symp.*, pp. 39-47, Oct. 1992.
[2] M. Litzkow and M. Solomon, "Supporting Checkpointing and Process Migration outside the Unix Kernel," *Proc. Usenix Winter 1992 Technical Conf.*, pp. 283-290, Jan. 1992.
[3] Y.-M. Wang and W.K. Fuchs, "Scheduling Message Processing for Reducing Rollback Propagation," *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 204-211, July 1992.
[4] J. Xu and R.H.B. Netzer, "Adaptive Independent Checkpointing for Reducing Rollback Propagation," *Proc. IEEE Parallel and Distributed Processing Symp.*, pp. 754-761, Dec. 1993.
[5] C.-C.J. Li, E.M. Stewart, and W.K. Fuchs, "Compiler-Assisted Full Checkpointing," *Software—Practice and Experience*, vol. 24, no. 10, pp. 871-886, Oct. 1994.
[6] J.S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," *Proc. Usenix Winter Technical Conf.*, pp. 213-223, Jan. 1995.
[7] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala, "Checkpointing and Its Applications," *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 22-31, June 1995.
[8] N. Neves and W.K. Fuchs, "Using Time to Improve the Performance of Coordinated Checkpointing," *Proc. IEEE Int'l Computer Performance & Dependability Symp.*, pp. 282-291, Sept. 1996.
[9] G. Cao and M. Singhal, "Low-Cost Checkpointing with Mutable Checkpoints in Mobile Computing Systems," *Proc. 18th Int'l Conf. Distributed Computing Systems*, pp. 464-471, May 1998.
[10] K.F. Ssu, B. Yao, W.K. Fuchs, and N. Neves, "Adaptive Checkpointing with Storage Management for Mobile Environments," *IEEE Trans. Reliability*, vol. 48, no. 4, pp. 315-324, Dec. 1999.
[11] B. Yao, K.F. Ssu, and W.K. Fuchs, "Message Logging in Mobile Computing," *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 294-301, June 1999.
[12] C.Y. Lin, S.Y. Kuo, and Y. Huang, "A Checkpointing Tools for Palm Operating System," *Proc. Int'l Conf. Dependable Systems and Networks*, pp. 71-76, July 2001.
[13] Y. Hollander and G.M. Silberman, "A Mechanism for the Migration of Tasks in Heterogeneous Distributed Processing Systems," *Proc. Int'l Conf. Parallel Processing and Applications*, pp. 93-98, Sept. 1988.
[14] M.M. Theimer and B. Hayes, "Heterogeneous Process Migration by Recompilation," *Proc. 11th Int'l Conf. Distributed Computing Systems*, pp. 18-25, July 1991.
[15] A. Beguelin, E. Seligman, and P. Stephan, "Application Level Fault Tolerance in Heterogeneous Networks of Workstations," Technical Report CMU-CS-96-157, Carnegie Mellon Univ., Aug. 1996.
[16] P.E. Chung, Y. Huang, S. Yajnik, G. Fowler, K.-P. Vo, and Y.-M. Wang, "Checkpointing in CosMiC: A User-Level Process Migration Environment," *Proc. Pacific Rim Int'l Symp. Fault-Tolerant Systems*, pp. 187-193, Dec. 1997.
[17] B. Ramkumar and V. Strumpen, "Portable Checkpointing for Heterogeneous Architectures," *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 58-67, June 1997.
[18] F. Karablieh, R.A. Bazzi, and M. Hicks, "Compiler-Assisted Heterogeneous Checkpointing," *Proc. IEEE Reliable Distributed Systems Symp.*, pp. 56-65, Oct. 2001.
[19] R.K.K. Ma, C.-L. Wang, and F.C.M. Lau, "M-JavaMPI: A Java-MPI Binding with Process Migration Support," *Proc. Int'l Symp. Cluster Computing and the Grid*, pp. 240-247, May 2002.
[20] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.
[21] C.-C.J. Li and W.K. Fuchs, "CATCH—Compiler-Assisted Techniques for Checkpointing," *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 74-81, June 1990.
[22] E. Seligman and A. Beguelin, "High-Level Fault Tolerance in Distributed Programs," Technical Report CMU-CS-94-223, Carnegie Mellon Univ., Dec. 1994.
[23] L. Alvisi and K. Marzullo, "Message Logging: Pessimistic, Optimistic, Causal, and Optimal," *IEEE Trans. Software Eng.*, vol. 24, no. 2, pp. 149-159, Feb. 1998.
[24] E. Elnozahy, D. Johnson, and Y.-M. Wang, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," Technical Report CMU-CS-96-181, Carnegie Mellon Univ., Oct. 1996.
[25] N. Neves and W.K. Fuchs, "RENEW: A Tool for Fast and Efficient Implementation of Checkpoint Protocols," *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 58-67, June 1998.
[26] N. Neves and W.K. Fuchs, "Fault Detection Using Hints from the Socket Layer," *Proc. IEEE Reliable Distributed Systems Symp.*, pp. 64-71, Oct. 1997.
[27] Standard Performance Evaluation Corp., http://www.specbench.org, 2002.
[28] Bench Web, http://www.netlib.org/benchweb/, 2002.



Kuo-Feng Ssu received the BS degree in computer science & information engineering from National Chiao Tung University and the PhD degree in computer science from the University of Illinois at Urbana-Champaign. He is an assistant professor in the Department of Electrical Engineering, National Cheng Kung University, Taiwan. His research interests include dependable systems, mobile computing, and distributed systems. He is a member of the IEEE and the Phi Tau Phi honor scholastic society.

W. Kent Fuchs received the BSE degree from Duke University, the MDiv degree from Trinity Evangelical Divinity School, and the PhD degree in electrical engineering from the University of Illinois. He is dean of the College of Engineering, Cornell University. He was formerly head of the School of Electrical and Computer Engineering, Purdue University, and Michael J. and Catherine R. Birck Distinguished Professor. Before serving at Purdue, he was a professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois. His research interests include dependable computing, testing, and failure diagnosis. Research awards include the Senior Xerox Faculty Award for Excellence in Research, selection as a University Scholar, appointment as a fellow in the Center for Advanced Studies, and the Xerox Faculty Award for Excellence in Research, all from the University of Illinois. He received the Best Paper Award IEEE/ACM Design Automation Conference and the Best Paper Award IEEE VLSI Test Symposium. Dr. Fuchs has been a guest editor of special issues for *IEEE Transactions on Computers* and *Computer*. He has been a member of the editorial board for the *IEEE Transactions on Computers*, the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, and the *Journal of Electronic Testing: Theory and Applications*. He is a fellow of the IEEE and a fellow of the ACM.

Hewijin Christine Jiau received the BE degree in electrical engineering from National Cheng Kung University, the MS degree in electrical engineering & computer science from Northwestern University, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign. She is an assistant professor in the Department of Electrical Engineering, National Cheng Kung University, Taiwan. Her research interests include software reuse, object technologies, information integration, data mining, and database applications on the Internet. She is a member of the IEEE.