TECHNIQUES FOR MIGRATING COMPUTATIONS ON THE GRID*

VIPIN CHAUDHARY† AND HAI JIANG‡

Abstract. Grid computing is focusing more on resource sharing, cycle stealing, and other modes of collaboration among dynamic and geographically distributed organizations. As a complement to the traditional data migration in client/server and distributed computing systems, moving computations to proper locations for resource sharing becomes an effective alternative. The Grid's characteristics require computation migration schemes be able to fit the underlying dynamic and heterogeneous computing environments. Although no fully-functioned such system is available, current migration systems have achieved partial success at different aspects. Major difficulties and issues are identified, and thereafter possible solutions are proposed. Especially, a multi-grained computation migration system, MigThread, is outlined to demonstrate the feasibility of moving computations in Grid-like environments. Possible techniques are discussed to shed light on the system design of next generation computation migration which is the essence to enable flexible resource sharing on the Grid.

Key words. process migration, thread migration, mobile agent, checkpointing, Grid computing

AMS subject classifications. 68M14, 68M15, 68M20

1. Introduction.

From cluster computing to internet computing and now Grid computing, current computation technologies have focused more on collaboration, data sharing, cycle stealing, and other modes of interaction among dynamic and geographically distributed organizations [1]. At the heart of the Grid is the ability to discover, allocate, and negotiate the use of system resources, such as a computer, network, or storage system.

Resource management in traditional computing systems is a well-studied problem. Resource managers exist in many computing environments and include bath schedulers, task brokers, workflow engines, and operating systems. The representative systems include Condor [2], LSF [3], LoadLeveler [4], and NQE [5] spread throughout academia and business. However, all of them can only work smoothly within their clusters and local organizations. What distinguishes resource management in the Grid environment is the fact that the managed resources span administrative domains [6]. New protocols and APIs are required to collect local resource managers together and enable them work well in Grid environments. The Globus Toolkit's Grid Resource Allocation Manager (GRAM) is representative of first-generation Grid resource management systems [7] which interfaces to local resource managers and relies on them to provide actual functions. Once all local resource managers become Grid-enabled, such as Condor-G [8], resource utilization can be arranged Grid-wide.

However, most resource management systems can only handle batch jobs. First, they detect available local or remote resources. And then a certain computation job is assigned to a selected resource. Normally the job will stay at the resource location until it finishes the execution. This is a typical static scheduling scenario. On the Grid, the ability of re-arranging running computation jobs is on demand since such dynamic rescheduling can exploit newly detected resources efficiently at runtime, especially for those long-running scientific applications. Computation mobility can enable job reconfiguration on the fly to fit dynamically changing Grid environments. In

*This research was supported in part by NSF IGERT grant 9987598, NSF MRI grant 9977815, NSF ITR grant 0081696.

†Institute for Scientific Computing, Wayne State University, Detroit, MI (vipin@wayne.edu).

‡Department of Computer Science, Arkansas State University (hjiang@csm.astate.edu).

fact, job/computation migration improves both application performance and system resource utilization efficiency. It has become indispensable for load balancing, load sharing, fault tolerance and data locality improvement.

Load balancing concerns even distribution of workload over multiple processors to improve whole parallel computations' performance. For scientific applications, computations are partitioned into multiple tasks running on different processors/computers. In addition to different computing powers, multiple users and multiple programs share the computation resources in the non-dedicated computing environments, such as Grids. For parallel applications, load imbalanced scenario occurs frequently even though the workload was distributed evenly before. Therefore, dynamically and periodically adjusting workload distribution is required to make sure that all running tasks at different locations will finish their execution at the virtually same time, minimizing the idling time. Such load reconfiguration needs to migrate tasks from one location to another.

From the system's point of view, load sharing will typically increase the throughput of Grids. Studies have indicated that a large fraction of workstations could be unused for a large fraction of time [9]. Grid computing seeks to exploit otherwise idle workstations and PCs to create powerful distributed computing systems with global reach and supercomputer capabilities [10]. With the migration capability, a systematic scattering of computations across processors which allows heavily loaded processors to efficiently balance their load with lightly loaded processors gives the executing Grid an opportunity to achieve a better overall throughput.

Sharing resources includes two approaches: moving data to computation or moving computation to data. Current applications favor data migration as in FTP, web, and Distributed Shared Memory (DSM) systems. However, when computation sizes are much smaller than data sizes, code or computation migration might be more efficient. Communication frequency and volume will be minimized by converting remote accesses into local ones. In data intensive computing, when client-server and RPC infrastructure are not available, computation migration is an effective alternate to access massive remote data.

In this book chapter, computation migration issues and techniques are identified to shed light on the deployment of migration packages in Grids. The remainder of chapter is organized as follows: Section 2 introduces different computation migration strategies. In Section 3, migration system's design issues and technologies are described in details. Section 4 is the case study of a multi-grained computation migration package, *MigThread*, which is the prototype implementation of a resource utilization system in Grids. Finally, the conclusions and future work are presented in Section 5.

2. Computation Mobility. The notion of *computation mobility* concerns that a computation starting at some network node may continue execution at some other network node in Grids. It involves much more than just moving code. In fact, it moves both code and execution contexts. How to construct and reload the execution contexts at runtime is the essence of *computation mobility*.

Sometimes data is bound closely with computations. Then, *data mobility* might imply *computation mobility* by moving both data and computations. In fact, this indicates the *data-computes rule*: computation works only on its own data.

Another similar notation is *control mobility*: a thread of control originating at some network node continues execution at some other network node, and then comes back. This is the scenario in the Remote Procedure Call (RPC) model [11]. No code

is moved in this process, just control although data mobility is also involved.

Systems with *computation mobility* are classified by the granularity and location of the execution context or its replica. A variety of research activities have been reported in the literature.

2.1. Process Migration. A process is an operating system abstraction representing an instance of a running computer program. Each process contains its own address space, program counter, stack, and heaps. Processes can spawn child processes and work together. However, this multi-process programming paradigm is not a popular approach anymore because of the heavy-weight context switch between processes. Most of the time, a process represents the whole computing unit no matter it is a sequential computation or a complete parallel computation.

Process migration concerns migrating the running program from one location to another for a typical "all or nothing" scenario. Since the process is the computing unit, it is often called coarse-grained computation migration. The representative systems with process migration functionality include Mosix [12], Sprite [13], Mach [14], V system [15], Condor [20], Tui [21], Process Introspection [22], MigThread [23], and SNOW [24].

The major issue in process migration is how to fetch the accurate process state. To obtain the original state in operation systems, kernel is normally modified since no proper system calls are provided. Also, the portability is a major issue since process states are operating system abstractions. This prevents most process migration systems from being used widespread on the Grid. The alternative is to duplicate process states in user space. Moving states up to higher levels enables the heterogeneous migration. Most recent system put their focus on this [23, 21, 24].

2.2. Thread Migration. Threads (lightweight processes) are flows of control in running programs. One process might contain multiple threads which share the same address space including text and data segments. However, each thread may have its own stack and heap. Threads can represent a mount of computations and be treated as a computing unit. Then, a process can be further decomposed into multiple smaller computing tasks in the format of threads.

Thread migration enables fine-grained computation adjustment in parallel computing. As multi-threading becomes a popular programming practice, thread migration is increasingly important in fine-tuning the high performance computing or Grid computing to fit dynamic and non-dedicated environments. Different threads can be migrated to utilize different resources for load balancing and load sharing. The core of thread migration is about how to transfer thread state and necessary data in local heap to the destination.

Current thread migration research focuses on updating internal self-referential pointers in stacks and heaps. The first approach uses language and compiler support to maintain enough type information and identify pointers as in MigThread [23] and Arachne [26]. The second approach requires scanning the stacks at run-time to detect and translate the possible pointers dynamically. The representative implementation of this is Ariadne[27]. Since some pointers in stack cannot possibly be detected (as pointed out by [31]), the resumed execution can be incorrect. The third approach is most popular, and necessitates the partitioning of the address space and reservation of unique virtual address for the stack of each thread so that the internal pointers maintain the same values. A common solution is to preallocate memory space for threads on all machines and restrict each thread to migrate to its corresponding location on other machines. This "iso-address" solution requires large address space

and is not scalable since there are limitations on stacks and heaps [31]. Such systems include Amber [28], UPVM system [29], PM2 [30], Millipede [31], Nomad system [32] and the one proposed by Cronk et. al. [33].

Based on the location of definition, threads can be classified as kernel, user, and language level threads. Kernel level threads exist in operating systems and can be scheduled onto processors directly. User level threads are defined and schedule by libraries in user space. Language level threads are defined in language. For example, Java thread is defined in Java language and implemented in Java Virtual Machine (JVM). According to the thread type, migration systems have to fetch thread states from different places and port them to different platforms. Up to now, only MigThread [23] and Jessica2 [51] support heterogeneous thread migration. MigThread achieves this by defining its own data conversion scheme whereas Jessica2 relies on modified JVMs.

2.3. Checkpointing. Checkpointing provides the backbone for rollback recovery (fault-tolerance), playback debugging, and job swapping. Checkpointing is the saving of computation state, usually to stable storage, so that it may be reconstructed later in time. Therefore, the major difference between migration and checkpointing is the medium: memory-to-memory vs. memory-to-file transfer. Checkpointing may apply most migration strategies. Dr. James Plank conducted a research project at the University of Tennessee to develop architecture-independent checkpointing (AIC) for Java programs [34]. It modifies JVM "Kaffe" using the mark and sweep code for garbage collector to save the state of each object.

Libckpt [35], PREACHES [36], Porch [37], CosMic [38], and other user-directed checkpointing systems [39, 40] save process states into stable storage, such as magnetic disks. Memory exclusion technique has been employed effectively in incremental checkpointing, where pages are not checkpointed when they are clean. In other words, their values have not been altered since the previous checkpoint [41]. The next step, "compiler-assisted checkpointing," uses compiler/preprocessor to ensure correct memory exclusion calls for better performance [42].

For message passing and shared address space parallel computing applications, CoCheck [43] and C^3 [44, 45] manage to get clear-cut checkpoints which can be treated as computation states for migration. Hence, from the computation state's point of view, migration and checkpointing systems are exchangeable.

2.4. Virtual Machines. To enable code mobility in heterogeneous environments, Virtual Machine (VM) technique is widely used, for example in JVM [46], PVM [47], and VMware [48]. It presents the image of a dedicated raw machine to each user [49]. Virtual machines allow the configuration of an entire operating system to be independent from that of the physical resource; it is possible to completely represent a VM "gust" machine by its virtual state and instantiate it in any VM "host" [50]. Furthermore, VMs can be migrated to appropriate locations for resources. Most of the time, this heavy-weighted operating system image might cause tremendous migration overhead. Those resources and devices not used in the current computation should not be included in the image file. However, VMs always have difficulties to distinguish which resources are useful or not. The safe way is to wrap up the whole abstract view of the underlying physical machine. Then efficiency drops dramatically.

Although migrating VM itself is not a practical solution for run-time migration, some process/thread migration systems, such as SNOW [24] and Jessica2 [51], work on top of VMs to enable the migration in heterogeneous environments. VMs are used to interpret computation states to hide low-level architecture variety. VMs play a role

as data converter [24]. The second approach is to represent computation in VMs, i.e., computation states are in VMs. Since VMs provide uniform platforms, states can be fetched in a unique way and heterogeneity issue is resolved smoothly [51].

2.5. Mobile Agents. An mobile agent is a software object and represents an autonomous computation that can travel in a network to perform tasks on behalf of its creator. It has the ability to interact with its execution environment, and to act asynchronously and autonomously upon it. Mobile agents reduce the network load and overcome the network latency. They travel with their code, data, and states. The states determine what to do when they resume execution at their destinations. The code is their object-oriented context, and most existing mobile agent systems, including Telescript [52], IBM Aglets [53], Agent Tcl [54], and Neplet [55], implement their agents in object-oriented languages such as Java and Telescript.

Mobile agents demand a different whole coding environment, including new language constructs, programming styles, compilers, and execution platforms. Although current mobile agent systems are intended for general applications and have demonstrated some progress in internet/mobile computing, it is still not clear how they will perform for computation-intensive and high performance computing applications with object-oriented technology. And legacy systems will have to be re-written to accommodate agents. Thus, this technique has not been accepted widely in high performance/parallel computing.

3. Design Issues and Technologies. Different computation migration systems adopts variant approaches to pause computations, construct computation states, transfer computations to destinations, and resume computations remotely. Although these systems are designed and deployed in local clusters, they can be extended to Grids easily by applying the Grid protocols and APIs. Grid resource managers do not involve in the computation restructuring and transfer. All techniques used in clusters are reusable. However, they need to be reshuffled properly to fit in the new Grid infrastructure. The major obstacle preventing computation migration from achieving widespread use is the complexity of adding transparent migration to systems originally designed to run stand-alone [56]. Transparency, applicability, and overheads are the major concerns in system design.

3.1. Granularity. The term granularity refers the size of computation units which can move around individually. Normally it indicates the flexibility of mobility systems can provide.

When Virtual Machines' whole system images are dumped as migration units, all computations over the VM will be transferred together and they cannot be distinguished explicitly. Sequential and parallel jobs are treated the same. This extreme VM migration case is efficient only when all local jobs need to leave the current machine. User programs are untouched while mobility is available even in heterogeneous environments. Application programmers are not aware of the migration, but high overheads and inflexibility are obvious.

From operating systems' point of view, processes are the basic computation abstraction. All sequential computations contain just single processes. In parallel computing, the overall jobs need to be decomposed into multiple tasks. In multi-process parallel applications such as those using MPI (Message Passing Interface), tasks are assigned to processes for parallel execution. In such cases, processes are treated as partial computations. Compared to VM migration, process migration shows its advantages by reducing overheads significantly and manipulating individual or even

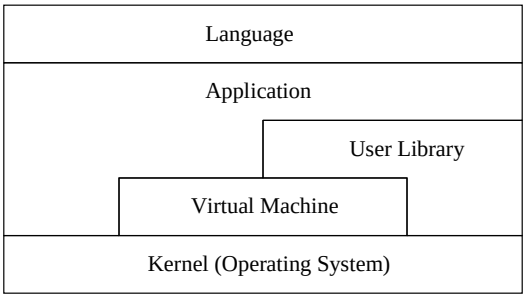

FIG. 3.1. *Levels of Migration Implementation.*

parts of applications. Some parallel applications can be reconfigured on the fly. Different implementations of process migration provide various transparency levels to programmers.

To reduce the heavy inter-process context switch overhead, many modern parallel computing applications adopt multi-threading technique. Each process may contain multiple threads to share the same address space. Former multi-processed applications can be replaced by multi-threaded counterparts. Or former processes are partitioned further into more local threads to achieve finer task decomposition. More parallelism will be exploited and performance will be improved as well. Sequential applications can be viewed as single-thread instances while parallel ones consist of multiple threads. Once threads are used as computation units, thread migration can move sequential jobs and entire or partial parallel jobs. Since process migration treats multi-threaded applications as a whole, it will either move the whole computation or activate no migration. Such "all-or-nothing" scenarios do not exist in thread migration. However, some thread libraries are invisible to operating systems so that the difficulty of implementing thread migration is higher than process migration.

Charm++ [57], Emerald [25], and other mobile agent systems provide mechanisms at languages level to migrate the user defined objects/agents. The sizes of computation units vary according to the applications. However, new languages and compilers expose everything to programmers and legacy systems have to be re-deployed. Transparency is the major drawback in this approach.

Granularity selection is based on applications. To achieve high transparency and low overhead, thread migration seems the good candidate for legacy and high performance computing applications whereas mobile agents performs well in mobile and internet computing.

3.2. Implementation Levels. Migration systems can be implemented at different levels as shown in Fig. 3.1. Different approaches demonstrate various results of transparency and flexibility.

3.2.1. Kernel Level. Kernel-level approach is only adopted by process migration since other computation unit definitions are out of the control scope of operating systems. For example, threads are classified as kernel, user, and language threads. Kernel can only detect kernel threads. However, applications can only use user threads (defined in user libraries) or language threads like Java thread. In process state image, kernel cannot tell which portion belongs to high-level threads. Then migrating threads becomes impossible.

Process migration functionality has been implemented in some distributed or networked operating systems, such as MOSIX [12], Sprite [13], Mach [14], Accent [16], V

[15], Locus [17], and OSF1 AS TNC [18]. These operating systems can access process state efficiently and support preemptive migration at virtually any point. Thus, they provide good transparency and flexibility to end users. However, this approach brings much complexity into kernels and process images cannot be shared between different operating systems. Therefore, the single system image requirement forces both source and destination nodes to be under control of the same operating systems, and such model is unacceptable in common distributed systems. Some systems such as MOSIX, Sprite and Mach migrate the whole process images whereas others like Accent and V Kernel apply "Copy-On-Reference" and "precopying" techniques to shorten the process freeze time. The Beowulf Distributed Process Space (BProc) [19] is set of kernel modifications, utilities and libraries which allows a user to start processes on other machines in a Beowulf-style cluster as well. Heterogeneity is the drawback of this approach since process images can only remain unchanged in the same operating system environment. On the Grid, computation resources spread across multiple administrative domains. Same operating systems and versions cannot be guaranteed.

3.2.2. Virtual Machine Level. VM-level migration is good at supporting heterogeneity. Virtual machines provide abstract operating system which is independent from the physical resources. Computation can migrate across different platforms. Because of heavy-weight computation image and coarse-grained migration unit definition, VM migration is not widespread used. Only if computations are decomposed at levels above VM, could the system performance be promising.

3.2.3. User Level. The user-level migration reduces the complexity in modifying kernels. Normally user libraries are used and linked to the application at compile time. Only process migration can adopt this approach and computation state image is constructed through library calls. The migration time is typically a function of the address space size, since the eager data transfer scheme is deployed. Condor [2] and Libckpt [35] are good examples of this approach. No kernel modification is required and this approach can achieve almost the same result as kernel-level migration. Since system calls are invoked in library calls to fetch process images from the kernel, user-level migration still only works on homogeneous platforms. The only big progress is to eliminate kernel modification.

3.2.4. Application Level. The migration can also be implemented as part of an application. Such an approach deliberately sacrifices transparency and reusability although precompilers/preprocessors are usually required to improve transparency in certain degree. Both process and thread migration can be implemented at this level. The computation state can also move into user space to support migration in heterogeneous environments. The Tui system [21] provides heterogeneous process migration by modifying the ACK (Amsterdam Compiler Kit). MigThread [23], SNOW [24], Porch [37], PREACHES [36] and Process Introspection [22] support heterogeneous process migration and apply source-to-source transformation to convert programs into semantically equivalent source programs for saving and recovering state across binary incompatible machines. MigThread supports thread migration as well. Jessica2 [51] provides Java thread migration by modifying Java Virtual Machine (JVM) for Java thread states. JVM is also useful for heterogeneous migration. However, JVM has to be distributed before computations.

If the precompiler/preprocessor is strong enough to provide sufficient transparency and granularity is reduced to threads, application-level migration will pose good potential in Grids.

3.2.5. Language Level. Flexible granularity makes the language-level migration attractive. At such a high level, flexibility and heterogeneity are guaranteed. Programmers can define computation units as in mobile agent systems [52, 53, 54, 55]. Charm++ is a machine independent parallel programming system and requires a new compiler [57]. Emerald is a programming language and an environment that supports mobile objects in a distributed environment [25]. New languages, compilers, and programming styles turn into obstacles since programmers have to give up what they have been familiar with for a long time to adopt the new replacements. Legacy systems are left behind, and how to handle high performance computing is a research topic.

3.3. Heterogeneity. Heterogeneity is one of basic factors that distinguish Grid computing from traditional cluster computing. Migration systems which only work on homogeneous platforms will face challenges in Grids. Both kernel-level and user-level migration belongs to this category. While large overheads hassle VM migration, application-level and language-level migration becomes the only choice.

Most heterogeneous language-level migration systems including mobile agents rely on virtual machines to tolerate different platforms. Hybrid-level systems like Jessica2 [51] is also bonded with VM. Before a particular VM dominates Grid environments, all these systems will not be widespread used.

Application-level approaches without VM place few assumption in heterogeneous Grid environments. Normally they are equipped with their own solutions. The computation state is represented in pure data format. In order to make these states acceptable across various platforms, data conversion standards are usually adopted.

Most data conversion schemes, such as XDR (External Data Representation) [58], adopt a canonical intermediate form strategy which provides an external representation for each data type. This approach requires the sender to convert data to canonical form and the receiver to convert data from canonical form. Even if both the sender and receiver are on the same machine, they still need to perform this symmetric conversion on both ends. Obviously, computational overhead is high, especially for large data chunks. XDR is one of the most popular representations in this group. It adopts the untagged data representation approach (except for array lengths) and encodes only the data instead of data types. Therefore, data types have to be determined by application protocols. For each data type, there are corresponding routines for its encoding and decoding. And each platform is only equipped with a set of conversion routines against this machine-independent format.

Zhou and Geist [59] proposed an asymmetric data conversion technique, called "receiver makes it right" (RMR), where the sender sends data in its own representation and data conversion is performed only on the receiver side. Thus, the receiver should be able to convert and accept data from all other machines. Just as in XDR, RMR typically flattens complex data structure and associates packing/unpacking routines with each basic data type. If there are n kinds of different machines, the number of conversion routine groups will be $(n^2 - n)/2$. In theory, the RMR scheme will lead to bloated code as n increases.

Both the above symmetric and asymmetric conversion strategies require flattening complex data structures and associating packing/unpacking routines with each basic data type because the padding patterns in aggregate types are a consequence of the processor, operating system and compiler, and cannot be determined until run-time. This data type flattening process incurs tremendous coding burden for programmers.

Some process migration systems can work in heterogeneous environment. Only

MigThread [60] and Jessica2 [51] are known for supporting heterogeneous thread migration. The Tui system [21] defined its own symmetric data conversion scheme. SNOW [24] adopted XDR and Process Introspection [22] used RMR. MigThread [23] proposed a "plug-and-play" style data conversion scheme, called Coarse-Grain Tagged "receiver makes it right" (CGT-RMR) [60], which is an asymmetric data conversion method to perform data conversion only on the receiver side. This tagged RMR version scheme can tackle data alignment and padding physically, convert data structures as a whole, and eventually generate a lighter workload compared to existing standards.

3.4. Migration Safety. Migration safety concerns precise state construction for successful migration. VM migration uses VM abstraction to guarantee the correctness of interpretation. Kernel-level and user-level migration only works on homogeneous platforms and process state images are unchanged. Language-level migration systems define computation states at language level and are often equipped with VM support. Only the application-level approach requires to ensure state correctness, especially the heterogeneous migration.

Since the computation state consists of stacks and heaps, heterogeneous migration schemes need to interpret all memory segments precisely. They usually have difficulties in dealing with programs written in type-unsafe languages, such as C [21], where memory blocks' dynamic types could be totally different from the types declared in the program.

Type-safe languages can avoid such type uncertainty. That is, static types declared within a program will be the same as the dynamic types at run-time. No type casting is allowed, and no type change occurs once programs start running. Thus, migration/checkpointing schemes can interpret contents of memory segments accurately and build thread states correctly. But "type-safe languages" restriction is too conservative since many programs written in such languages might be safe for migration. Most schemes rely on type-safe programming styles [21, 24]. It is impractical to rely on programmers to identify unsafe language features and avoid using them. Major unsafe uses come from pointers and their operations. If a pointer is cast into an integral-type variable before migration/checkpointing and cast back to a pointer variable later, a scheme might fail to identify the real data type in the memory block statically declared as an integral type, and miss the pointer updating procedure during the migration. Then, subsequent use of the pointer with invalid values can lead to errors or incorrect results. Therefore, events generating hidden pointers such as pointer casting are the actions we must forbid. MigThread applied a pointer inference algorithm [61] to detect hidden pointers. Then programmers are free to write code in any programming style.

Another unsafe factor is third-party library calls which may leave undetectable memory blocks and pointers out of the control scope. Without precise understanding, migration schemes will fail to determine their existence. MigThread can issue compile time warnings of the potential risks [62].

3.5. Adaptation Points. Migration points reflect the adaptability of computations. Mobile agents can interact with environments and make travel decision themselves. However, normally computations will receive commands passively from Grid schedulers or resource managers. Systems using kernel-level, user-level, and VM migration can accept migration request at anytime. Computation images/states will be dumped instantly. Contents of address space and hardware registers are saved for migration.

Application-level approaches need to reconstruct computation states in the applications and only allow such events to take place at certain convenient positions, called adaptation points. Applications employ polling strategy to check migration commands at these adaptation points which are the superset of migration points. The locations and types of adaptation points are critical since the distance between two consecutive adaptation points determines the migration scheme's sensitivity and overheads. If they are too far apart, applications might be insensitive to the dynamic situation. But if they are too close, the related overheads will slow down the actual computation. Most migration systems do not elaborate this issue. SNOW [24] suggested to insert adaptation points based on the number of floating point operations although how to count operations in irregular problem applications is unclear. MigThread [23] proposed a heuristic algorithm to insert adaptation points based on programming constructs such as loops and function calls [63]. The polling overhead is reduced to small enough so that more adaptation points may not incur any significant slowdown.

In parallel applications, sometimes adaptation points can be only inserted at the locations where all parallel parts are in consistent status. Synchronization points are good candidates since data copies are synchronized and no transient messages are on the fly. MigThread inserts *pseudo-barriers* to only synchronize task progress for adaptation points [63]. C^3 [44, 45] proposed non-synchronization schemes to checkpoint shared-address-space and message-passing programs. Same strategy can be also applied to migration case and will be essential to support adaptive parallel computing.

3.6. Communication and I/O support. After migration, how computations can continue previous communication and issue new I/O operations to original hosts is still a big challenge in most current migration systems. Only kernel-level migration can resolve this seamlessly since networked and distributed operating systems have full control on both source and destination machines. Reconnecting communication and forwarding I/O requests back to source nodes are straightforward since communication protocols and I/O device access are all under operating systems' control.

Without kernel's support, user-level and application-level migration systems have to catch communication and I/O requests before issuing system calls to kernels. Condor [2] proposed a *shadow process* concept to leave the old precess at the source machines. The *shadow processes* keep the old communication channels and forward newly arrived messages to the new processes on the destination machines. After migration, I/O requests for file read/write, signals, and device accesses issued by resumed processes will be sent back to *shadow processes* and performed on the original machines. This is a generic strategy which can handle the communications in open systems, i.e., the other end of the original communication channels can be unknown. But the communication primitive syntax has to be well defined so that messages can be forwarded back and forth in *shadow processes*.

In closed systems, communication problem is relatively easier to deal with because migration systems have full control of communication's senders and receivers. Communication primitives are easily replaced or wrapped. SNOW [24] supports continuous communication in heterogeneous process migration with the support from PVM (Parallel Virtual Machine). Since both ends of communication are using PVM and under control, *send* and *receive* primitives are rewritten to contain reconnection functionality. Any command issued after migration will implicitly tear down old communication channel and set up a new one. Transient messages are forwarded to new locations and message sequences are maintained. The restriction is that the modified

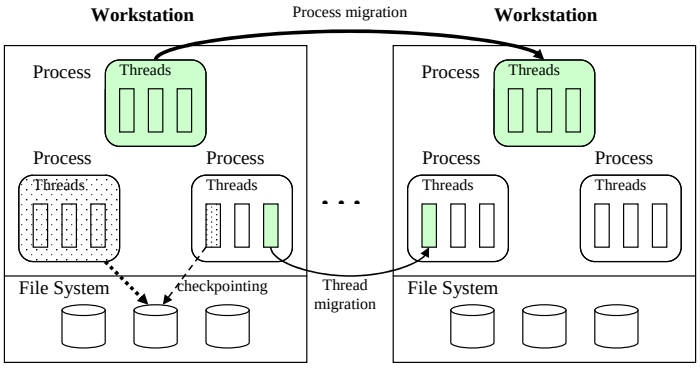

FIG. 4.1. **Process/thread and migration/checkpoing options in** *MigThread*.

version of PVM has to be installed ahead. Modifying communication primitives on other virtual machines or operating systems can achieve the same result. However, this strategy only works for closed systems. Open systems such as web servers using socket programming are not applicable.

4. Case Study: *MigThread*. *MigThread*, an application-level heterogeneous multi-grained computation migration and checkpointing system, is a representative infrastructure to demonstrate how to apply various effective migration strategies on the Grid [65]. Both coarse-grained processes and fine-grained threads are supported and both migration and checkpointing are available, as shown in Figure 4.1. For a certain process, its threads can simultaneously checkpoint to file systems or migrate to different destinations. This design brings sufficient flexibility into parallel computing. For process migration, all internal threads as well as their shared global data are processed together. Therefore, process migration could be the summation of the ones for all the internal threads.

4.1. State Construction. The state data typically consists of the process data segment, stack, heap and register contents. In *MigThread*, the computation state is moved out from its original location (libraries or kernels) and abstracted up to the language level. Therefore, the physical state is transformed into a logical form to achieve platform-independence and reduce migration restrictions, enabling *MigThread* not to rely on any type of thread library or operating system. Both the portability and the scalability of stack are improved. For threads, there is no need to preallocate addresses of stacks on each machine which is too costly and almost impossible in Grid computing environments. The limitation on virtual memory address space is also removed. User-level management of both stack and heap are provided. Pointers referencing stack or heap are represented in a generic manner so that they can be updated precisely after migration and checkpointing.

MigThread consists of two parts: a preprocessor and a run-time support module. The preprocessor is designed to transform user's source code into a format from which the run-time support module can construct the computation state precisely and efficiently. Its power can improve the transparency drawback in application-level schemes. The run-time support module constructs, transfers, and restores computation state dynamically as well as provides other run-time safety checks.

Handling computation state is the core of migration. State construction is done by both the preprocessor and the run-time support module. At compile time, all infor-

mation related to stack variables, function parameters, program counters, and dynamically allocated memory regions, is collected into certain pre-defined data structures [23].

In order to reduce migration overheads, many systems [21, 24, 22] intend to determine variables for migration dynamically, i.e., only the modified variables will be migrated. However, for applications written in C language, it is almost impossible for the preprocessor/precompiler to detect which variables are touched because of pointer arithmetic and dereference operations. Tracing variable access at runtime is too costly and will slow down the whole execution. To ensure the correctness of migration, *MigThread* collects globally shared variables and user functions' local variables into predefined data structures. During the migration, computation states will be constructed quickly by merging aggregate data structures instead of selecting variables one-by-one. Since the address spaces of a thread could be different on source and destination machines, and stacks and heaps need to be re-created on destination machines, values of pointers referencing stacks or heaps might become invalid after migration. The preprocessor run-time support module work together to identify and mark pointers at the language level so that they can easily be traced and updated later.

The program counter (PC) is the memory address of the current execution point within a program. It indicates the starting point after migration. When the PC is moved up to the language level, it is represented as a series of **integer** values. In the transformed code, a **switch** statement is inserted to dispatch execution to each labeled point according to the value of PC. The **switch** and **goto** statements help control jump to resumption points quickly.

MigThread also supports user-level memory management for heaps. In application programs, when **malloc()** and **free()** are invoked to allocate and deallocate memory space, extra statements are appended to trace memory blocks correspondingly. Eventually, all computation state related contents, including stacks and heaps, are moved out to the user space and handled by *MigThread* directly. This strategy achieves the foundation for correct state retrieval and fulfills the pre-condition for portability.

4.2. The Data Conversion Scheme. Computation states can be abstracted to language level and transformed into pure data. If different platforms use different data formats, then the computation states constructed on one platform need to be interpreted by another. Thus data conversion is unavoidable.

MigThread adopts a new data conversion scheme, called Coarse-Grain Tagged "receiver makes it right" (CGT-RMR) [60]. This tagged RMR version scheme can tackle data alignment and padding physically, convert data structures as a whole, and eventually generate lighter workload and less programming complexity compared to existing standards such as XDR [58]. It accepts ASCII character sets, handles byte ordering, and adopts IEEE 754 floating-point standard because of its dominance in the market.

With *MigThread*, programmers do not need to worry about data formats. The preprocessor parses the source code, sets up type systems, transforms source code, and communicates with the run-time support module through inserted primitives. The preprocessor can analyze data types, flatten down aggregate types recursively, detect padding patterns, and define tags as in Figure 4.2. But the actual tag contents can be set only at run-time and they may not be the same on different platforms. Since all of the tedious tag definition work has been performed by the preprocessor, the programming style becomes extremely simple. Also, with global control, low-level issues

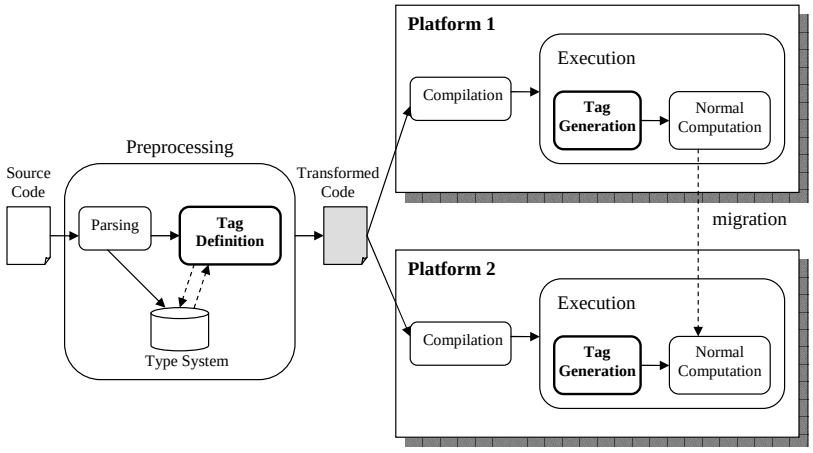

FIG. 4.2. **Tag definition and generation in** *MigThread*.

such as data conversion status can be conveyed to upper-level scheduling modules. Therefore, easy coding style and performance gains come from the preprocessor.

In *MigThread*, tags are used to describe data types and their paddings so that data conversion routines can handle aggregate types as well as common scalar types. Tags are defined and generated for these structures as well as dynamically allocated memory blocks in the heap. At compile time, it is still too early to determine the content of the tags. The preprocessor defines rules to calculate structure members' sizes and variant padding patterns, and inserts **sprintf()** to glue partial results together. The actual tag generation has to take place at run-time when the **sprintf()** statement is executed. Only one statement is issued for each data type, whether it is a scalar or aggregate type. The flattening procedure is accomplished by *MigThread*'s preprocessor during tag definition instead of the encoding/decoding process at run-time. Hence, programmers are freed from this responsibility.

In *MigThread*, all memory segments for predefined data structures are represented in a "tag-block" format. The process/thread stack becomes a sequence of these structures and their tags. Memory blocks in heaps are also associated with such tags to express the actual layout in memory space. Therefore, the computation state physically consists of a group of memory segments associated with their own tags in a "tag-segment" pair format.

CGT-RMR works in a "plug-and-play" style since it does not maintain tables or routine groups for all possible platforms as in traditional RMR [59]. No special requirement is imposed for porting it to a new platform.

4.3. State Restoration. On the sender side, there is no need to perform data conversion. But the receivers have to convert the computation state, i.e., data, as required. Since activation frames in stacks are re-run and heaps are recreated, a new set of segments in "tag-block" format is available on the new platform. *MigThread* first compares architecture tags by **strcmp()**. If they are identical and the blocks have the same sizes, the platforms remain unchanged and the old segment contents are simply copied over by **memcpy()** to the new architectures. This enables prompt processing between homogeneous platforms while symmetric conversion approaches still suffer data conversion overhead on both ends.

If platforms have been changed, conversion routines are applied on all memory

segments. For each segment, a "walk-through" process is conducted against its corresponding old segment from the previous platform. In these segments, according to their tags, memory blocks are viewed to consist of scalar type data and padding slots alternately. The high-level conversion unit is data slots rather than bytes in order to achieve portability. The "walk-through" process contains two index pointers pointing to a pair of matching scalar data slots in both blocks. The contents of the old data slots are converted and copied to the new data slots if byte ordering changes, and then the index pointers are moved down to the next slots. In the mean time, padding slots are skipped over. In *MigThread*, data items are expressed in "scalar type data - padding slots" pattern to support heterogeneity.

4.4. Safety Issues. *MigThread* can detect and handle some migration "unsafe" features, including pointer casting, pointers in unions, library calls, and incompatible data conversion. Then computation states will be precisely constructed to make those programs eligible for migration. Programmers are free to code in any programming style.

Pointer casting does not mean the cast between different pointer types, but the cast to/from integral types, such as integer, long, or double. The problem is that pointers might hide in integral type variables. The central issue is to detect those integral variables containing pointer values (or memory addresses) so that they could be updated during state restoration. Casting could be direct or indirect.

Pointer arithmetic and operations cause the harmful pointer casting, the most difficult safety issue. In *MigThread*, a novel intra-procedural, flow-insensitive, and context-insensitive pointer inference algorithm is proposed to detect hidden pointers created by unsafe pointer casting, regardless of whether it is applied in pointer assignments or **memcpy()** library call. This pointer inference algorithm has the following features:

- **Intra-procedure** : Analysis works within the function's scope. It does not consider the ripple effect of propagating unsafe pointer casting among functions.
- **Flow-insensitivity** : Control flow constructs are ignored. It is almost impossible to predict the exact execution order because of the potential dynamic inputs for the program. Conservative solutions might be safer for our case (pointer casting).
- **Context-insensitivity** : Different contexts do not be distinguished which simplifies the algorithm. For larger programs the exponential cost of context-sensitive algorithms quickly becomes prohibitive.

In C, the ways to modify memory contents include assignment statements, increment and decrement statements, and library calls. Since the increment and decrement statements only change data values instead of types, *MigThread* focuses on the other two. *MigThread* uses its type system to identify the types of the left-hand and right-hand sides of assignments. As type systems need to follow type inference rules, the pointer inference system has to apply pointer inference rules to detect unsafe pointer casting [61]. Pointer Casting (PC) Closure concept is used to denote an approximate superset of possible variables and locations holding hidden pointers.

The static analysis at compile time is the actual process that applies the pointer inference rules to the source code. It detects suspicious assignments and library calls, appends corresponding primitives to them for run-time checking, and transforms the source code. At run-time, the dynamic check section is activated by the primitives inserted at compile time. Clearly, if PC Closure is empty and no primitives are

inserted, the dynamic check will not be performed. During the dynamic check, another red-black tree is maintained to record actual unsafe pointer casting variables and locations. This is a precise set, unlike the approximate superset (closure) at compile time. And it may expand and shrink to reflect the real situation in program execution. With this algorithm, *MigThread* can either trace and recover unsafe pointer uses or simply abort the migration and checkpointing actions.

Union is a construct where pointers can evade updating since pointer fields' value can be referenced nonpointer fields. *MigThread* to trace the field use and identify the dynamic situation on the fly. In the program, once the preprocessor detects that a certain member of the union variable is in use, it inserts a primitive to inform the runtime support module which member and its corresponding pointer fields are in activation. The records for previous members' pointer subfields become invalid because of the ownership changing in the union variable. A linked list is used to maintain these inner pointers and get them updated after migration.

Library calls bring difficulties to all migration schemes since it is hard to figure out what is going on inside the library code. It is even harder for application-level schemes because they work on the source code and "memory leakage" might happen in the libraries. Without source code of libraries, it is difficult to catch all memory allocations because of the "blackbox" effect. The current version of *MigThread* provides a user interface to specify the syntax of certain library calls so that the preprocessor can know how to insert proper primitives for memory management and pointer tracing.

The final unsafe factor is the incompatible data conversion. Between incompatible platforms, if data items are converted from higher precision formats to lower precision formats, precision loss may occur. But if the high end portion of an unsigned number contains all-zero content, it is safe to throw them away since data values still remain unchanged. *MigThread* intends to convert data unless precision loss occurs. This aggressive data conversion enables more programs for migration without aborting them too conservatively. Detecting incompatible data formats and conveying this low-level information up to the scheduling module can help abort data restoration promptly.

In *MigThread*, pointer inference algorithm, a friendly user interface, and the data conversion scheme CGT-RMR work together to eliminate unsafe factors to qualify almost all programs for migration.

4.5. Adaptation Point Analysis. Since *MigThread* is an application-level migration system, certain code needs to be inserted into user programs in order to enable migration functionality. Instead of selecting actual adaptation points, *MigThread* proposed a strategy to aggressively insert potential adaptation points. Whether these adaptation points will be actually executed is determined by the scheduler which dynamically collects load information from all related machines. Once it determines that a thread/process needs to be migrated to another machine, it sends a signal to *MigThread* and the signal handler will set the migration flag so that the corresponding thread/process will be actually migrated at the next potential adaptation point. Since threads only check the true/false value of flag variables, they do not need to contact the scheduler directly and no further communication overhead will be introduced. Therefore, cost can be reduced significantly.

User programs consist of loops, common non-loop code blocks, function calls, and library calls. The preprocessor scans the program and detects all programming constructs. For non-nested loops, a potential adaptation point is inserted right after the last statement in the single loops. In case of nested loop, potential adaptation

points are inserted inside the innermost loop since the code in inner loop will also be executed in its outer loop.

Non-loop instructions including branch instructions will be ignored since they usually do not consume a lot of time as indicated in the experiments [63]. However, even if there are no loops in a subroutine, recursive functions and nested calls could consume a long period of time. So, at least one potential adaptation point is inserted at the end of the subroutine.

Library and system calls can cause problems to all application-level migration approaches since no source code is available to insert potential adaptation points at the high level. However, to achieve better portability of the application-level approach, it is reasonable to give up the sensitivity during the third-party library call procedures. Luckily, the execution time of most library calls is relatively short. System calls and I/O operations might take longer time according to the size of data bulk. This heuristic algorithm might suffer bad insensibility which declares a drawback in application-level approaches.

For some parallel applications where relaxed memory consistency models are used, pseudo-barrier primitive is inserted first to synchronize computation progress of multiple threads/processes across different machines. If any thread/process is scheduled for migration, real barrier operation will be activated to synchronize both computation progress and data copies. With little adaptation overhead, migration is achieved without any data consistency violation which is particularly critical in DSM systems [63].

MigThread is still underdevelopment. Many important features such I/O and communication operations are not included in the current version. How to support parallel computing on the Grid is not clear. However, this prototype demonstrates that many migration issues can be addressed and computation migration is feasible on the Grid.

5. Conclusions and Future Work. As resource sharing is one of the major goals of the Grid, data and computation movements are frequently on demand. Moving computations on the Grid requires a flexible and portable migration system to fit the dynamic and heterogeneous underlying environments. Existing systems still cannot provide a satisfactory solution, and this further prevents the Grid from being widely used. An efficient, practical, and flexible computation migration system is on demand.

This chapter identifies major issues in developing a mature computation migration system, including granularity, implementation levels, heterogeneity, migration safety, adaptation points, and communication and I/O support. The dominant factor is how to define a portable self-contained computation unit to fit the Grid. Since general applications are the target, fine-grained threads can be reasonably treated as the basic computation units to support both sequential and adaptive parallel computing. Application-level migration approach will be the only candidate in the heterogeneous Grid computing environments. Few restrictions may be placed on programs so that programmers can code in any style. The migration systems should be sensitive enough to dynamically load-changing environments. In addition, the communication and I/O need to be supported even in open systems.

Although no mature migration system exists on the Grid, he prototype system, *MigThread*, has indicated that multi-grained application-level migration approach is a good candidate and most issues can be resolved smoothly. However, communication and I/O operations in open systems are still under development. Current systems

can be used as guidelines to deploy the next generation migration system. Only when both data and computation migration schemes are ready, could resources be shared effective on the Grid. In another word, these migration techniques clean off the obstacles for Grid computing.

REFERENCES

[1] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, *Grid Services for Distributed System Integration*, Computer, 35(6) (2002).

[2] M. Litzkow and M. Livny, *Experience with the condor distributed batch system*, Proceedings of the IEEE Workshop on Experimental Distributed Systems (1990).

[3] S. Zhou, *LSF: Load sharing in large-scale heterogenous distributed systems*, Proceedings of the Workshop on ECluster Computing (1992).

[4] IBM Corporation, *IBM Load Leveler: User's Guide*, IBM Corporation (1993).

[5] Cray Inc., *Introducing NQE*, Technical Report 2153_2.97, Cray Inc., Seattle, WA (1997).

[6] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Second Edition, Morgan Kaufmann Publishers (2004).

[7] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, *A resource management architecture for metacomputing systems*, in 4th Workshop on Job Scheduling Stragegies for Parallel Processing, Springer-Verlag, Heidelberg (1998), pp. 62–82.

[8] J. Frey, T. Tannenbaum, I. Foster, M. Livny and S. Tuecke, *Condor-G: a computation management agent for multi-institutional grids*, Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (2001).

[9] J. K. Hollingsworth, P. J. Keleher and K. D. Ryu, *Resource-Aware Meta-Computing*, Advances in Computers, Vol 53, edited by Marvin V. Zelkowitz, Academic Press (2000).

[10] I. Foster, *Internet Computing and the Emerging Grid*, Nature Web Matters (2000).

[11] L. Cardelli, *Mobile Computations*, in Mobile Object Systems, edited by J. Viteck and C. Tschudin, Springer Verlag (1996).

[12] A. Barak and O. La'adan, *The MOSIX multicomputer operating system for high performance cluster computing*, Journal of Future Generation Computer Systems, 13(4-5):361-372 (1998).

[13] F. Douglis and J. K. Ousterhout, *Transparent process migration: Design alternatives and the Sprite implementation*, Software–Practice and Experience, 21(8):757-785 (1991).

[14] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, *Mach: A New Kernel Foundation for UNIX Development*, Proceedings of teh Summer USENIX Confernce, pp. 93-112 (1986).

[15] C. Shub, *Native Code Process-Originated Migration in a Heterogeneous Environment*, In Proceedings of the 1990 Computer Science Conference, pp. 266 - 270 (1990).

[16] R. Rashid and G. Robertson, *Accent: a Communication Oriented Network Operating SYstem Kernel*, Proc. of the 8th Symposium on Operating System Principles (1981).

[17] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, *The LOCUS Distributed Operating System*, Distributed Computing Systems: Concepts and Structures (1992).

[18] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala, *An OSF/1 UNIX for Massively Parallel Multicomputers*, Proc. of the Winter USENIX Conference (1993).

[19] E. Hendriks, *BProc: The Beowulf distributed process space*, Proc. of the 16th Annual ACM International Conference on Supercomputing (2002).

[20] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System* , Technical Report 1346, University of Wisconsin-Madison (1997).

[21] P. Smith and N. Hutchinson, *Heterogeneous process migration: the TUI system*, Software Practice and Experience, Vol. 28, no. 6 (1998).

[22] A. Ferrari, S. Chapin, and A. Grimshaw, *Process introspection: A checkpoint mechanism for high performance heterogenesous distributed systems*, Technical Report CS-96-15, University of Virginia, Department of Computer Science (1996).

[23] H. Jiang and V. Chaudhary, *Compile/Run-time Support for Thread Migration*, Proc. of ACM/IEEE International Parallel and Distributed Processing Symposium, pp. 58–66 (2002).

[24] K. Chanchio and X.H. Sun, *Data Collection and Restoration for Heterogeneous Process*

Migration, Proceedings of International Parallel and Distributed Processing Symposium (2001).

[25] E. Jul, H. Levy, N. Hutchinson, and A. Blad, *Fine-Grained Mobility in the Emerald System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, pp 109-133 (1998).

[26] B. Dimitrov and V. Rego, *Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms*, IEEE Transactions on Parallel and Distributed Systems, 9(5) (1998).

[27] E. Mascarenhas and V. Rego, *Ariadne: Architecture of a Portable Threads system supporting Mobile Processes*, Technical Report CSD-TR 95-017, CS, Purdue Univ. (1995).

[28] J. Chase, F. Amador, E. Lazowska, H. Levy and R. Littlefield, *The Amber System: Parallel Programming on a Network of Multiprocessors*, ACM Symposium on Operating System Principles (1989).

[29] J. Casa, R. Konuru, S. Otto, R. Prouty, and J. Walpole, *Adaptive Migration systems for PVM*, Supercomputing (1994).

[30] G. Antoniu and L. Boug, *DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols*, Proc. of 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (2001).

[31] A. Itzkovitz, A. Schuster, and L. Wolfovich, *Thread Migration and its Applications in Distributed Shared Memory Systems*, Journal of Systems and Software, vol. 42, no. 1 (1998).

[32] S. Milton, *Thread Migration in Distributed Memory Multicomputers*, Technical Report TR-CS-98-01, Department of Computer Sciences, The Australian National University (1998).

[33] D. Cronk, M. Haines, and P. Mehrotra, *Thread migration in the presence of pointers*, Proc. of the Mini-track on Multithreaded Systems, 30th Hawaii International Conference on System Sciences (1997).

[34] J. S. Plank, *Checkpointing JAVA*, http://www.cs.utk.edu/ plank/javackp.html.

[35] J. S. Plank, M. Beck, G. Kingsley and K. Li, *Libckpt: Transparent Checkpointing under Unix*, Proc. of Usenix Winter 1995 Technical Conference (1995).

[36] K. Ssu and W. K. Fuchs, *PREACHES - Portable Recovery and Checkpointing in Heterogeneous Systems*, Proc. of IEEE Fault-Tolerant Computing Symposium, pp. 38–47 (1998).

[37] B. Ramkumar and V. Strumpen, *Portable Checkpinting for Heterogeneous Architectures*, Proc. of 27th International Symposium on Fault-Tolerant Computing (1997).

[38] P. E. Chung et al, *Checkpointing in CosMiC: a user-level process migration environment*, Proc. of Pac. Rim Int. Symp. on Fault-Tol. Systems (1997).

[39] A. Agbaria and R. Friedman, *Virtual Machine Based Heterogeneous Checkpointing*, Proc. of 16th International Parallel and Distributed Processing Symposium (2002).

[40] K. F. Ssu, B. Yao, W. K. Fuchs, *An Adaptive Checkpointing Protocol to Bound Recovery Time with Message Logging*, Proc. of the 18th IEEE Symposium on Reliable Distributed Systems (1999).

[41] J. Plank, Y. Chen, K. Li, M. Beck and G. Kinsley, *Memory Exclusion: Optimizing the Performance of Checkpointing Systems*, Software – Practice and Experience, 29(2):125-142 (1999).

[42] J. S. Plank, M. Beck, and G. Kingsley, *Compiler-assisted memory exclusion for fast checkpointing*, IEEE Technical Committee on Operating Systems and Application Environments, 7(4):10–14 (1995).

[43] G. Stellner, *CoCheck: Checkpointing and Process Migration for MPI*, In Proc. of the 10th International Parallel Processing Symposium (1996).

[44] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, *Implementation and Evaluation of a Scalable Application-level Checkpoint-Recovery Scheme for MPI Programs*, Supercomputing (2004).

[45] G. Bronevetsky, D. Marques, M. Schulz, K. Pingali, and P. Stodghill, *Application-level Checkpointing for Shared Memory Programs*, Proc. of 11th International Conference on Architectural Support for Programming Languages and Operating Systems (2004).

[46] T. Lindholm and F. Yellin, *The Java(TM) Virtual Machine Specification*, Addison Wesley, 2nd edition (1999).

[47] V. S. Sunderam, *PVM: A framework for parallel distributed computing. Concurrency*, Concurrency: Practice and Experience, 2(4):315– 339 (1990).

[48] VMware Inc. *VMware virtual platform*, Technical white paper (1999).

[49] R. A. Meyer and L. H. Seawright, *A Virtual Machine Time Sharing System*, IBM System Journal, 9(3):199-218 (1970).

[50] R. Figueiredo, P. Dinda, and J. Fortes, *A Case For Grid Computing On Virtual Machine*, Proc. of 23rd International Conference on Distributed Computing Systems, pp. 550-559 (2003).

[51] W. Zhu, C.-L. Wang, and F. Lau, *Jessica2: a distributed Java virtual machine with transparent thread migration support*, Proc. of IEEE International Conference on Cluster Computing (2002).

[52] J. E. White, *Telescript Technology: Mobile Agents*, General Magic White Paper, J. Software Agents, AAI/MIT Press (1996).

[53] D. Lange and M. Oshima, *Mobile Agents with Java: The Aglet API*, Would Wide Web, 1(3) (1998).

[54] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko, *Agent Tcl: targeting the Needs of Mobile Computers*, IEEE Internet Computing, 1(4):58–67 (1997).

[55] C. Z. Xu, *Naplet: A Flexible Mobile Agent Framework for Network-Centric Applications*, Proc. of the Second Int'l Workshop on Internet Computing and E-Commerce (2002).

[56] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou, *Process Migration*, ACM Computing Surveys (2000).

[57] L. V. Kale and S. Krishnan, *Charm++: Parallel Programming with Message-Driven Objects*, Book Chapter in "Parallel Programming using C++", by Gregory V. Wilson and Paul Lu. MIT Press, pp. 175-213 (1996).

[58] R. Srinivasan, *XDR: External Data Representation Stndard*, RFC 1832, http://www.faqs.org/rfcs/rfc1832.html (1995).

[59] H. Zhou and A. Geist, *"Receiver Makes Right" Data Conversion in PVM*, In Proceedings of the 14th International Conference on Computers and Communications, pp. 458– 464 (1995).

[60] H. Jiang, V. Chaudhary, and J. Walters, *Data Conversion for Process/Thread Migration and Checkpointing*, Proc. of the International Conference on Parallel Processing (2003).

[61] H. Jiang and V. Chaudhary, *Thread Migration/Checkpointing for Type-Unsafe C Programs*, Proc. of ACM/IEEE International Conference on High Performance Computing, pp. 469-479 (2003).

[62] H. Jiang and V. Chaudhary, *On Improving Thread Migration: Safety and Performance*, Proc. of the International Conference on High Performance Computing (2002).

[63] Y. Ji, H. Jiang and V. Chaudhary, *Adaptation Point Analysis for Computation Migration/Checkpointing*, Technical Report, Institute for Scientific Computing, Wayne State University (2004).

[64] K. Chanchio and X. H. Sun, *Communication State transfer for the Mobility of Concurrent Heterogeneous Computing*, Proc. of the International Conference on Parallel Processing (2001).

[65] H. Jiang and V. Chaudhary, *Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems*, Proc. of the 37th Hawaii International Conference on System Sciences (2004).