Thread Migration/Checkpointing for Type-Unsafe C Programs[*]

Hai Jiang and Vipin Chaudhary

Institute for Scientific Computing
Wayne State University
Detroit, MI 48202 USA
{hai, vipin}@wayne.edu

Abstract. Thread migration/checkpointing is becoming indispensable for load balancing and fault tolerance in high performance computing applications, and its success depends on the migration/checkpointing-safety, which concerns the construction of an accurate computation state. This safety is hard to guard in some languages, such as the C, whose type-unsafe features may cause the dynamic types associated with memory segments to be totally different from the static types declared in programs. To overcome this, we propose a novel intra-procedural, flow-insensitive, and context-insensitive pointer inference algorithm to systematically detect and recover unsafe factors. The proposed scheme conducts a thorough static analysis at compile time and inserts a minimized set of primitives to reduce the overhead of possible dynamic checks at run-time. Consequently, most unsafe features will be removed. Programmers can then code in any style, and nearly every program will be qualified for migration/checkpointing.

1   Introduction

Thread migration and checkpointing has become an imperative means in redistributing computation [1] and fault-tolerance for parallel computing. All such schemes need to fetch (or construct), transfer, and re-install the computation state. The correct execution of this three-step procedure will guarantee correct results upon resuming work on a new machine. Since portable application-level schemes build the state from the source code, certain type unsafe features in C programs, such as pointer casting, structure casting with pointers, and pointer arithmetic, may cause the dynamic types associated with memory segments to be different from the static types declared in programs. In C, it is impossible to acquire the actual dynamic data types from the memory blocks themselves. This might prevent migration/checkpointing systems from getting the actual state and may result in migration/checkpointing failure. To avoid this, most schemes [3, 4] rely on type-safe programming styles.

We have designed and implemented a thread migration/checkpointing package, *MigThread* [5], to migrate threads safely for adaptive parallel computing. As an application-level approach, *MigThread* demonstrates good portability and scalability. Furthermore, we have identified pointer operations and library calls as the factors for unsafe migration/checkpointing [6]. To provide a reliable solution and cover all possible cases, we extend the previous work by proposing a novel intra-procedural, flow-insensitive, and context-insensitive pointer inference algorithm to detect unsafe pointer operations, and then systematically trace their propagation. To reduce the complexity, this algorithm focuses on C functions themselves, ignoring all flow control instruments and contexts. We argue that with the dynamic characteristic of programs, it is not always possible to predict the actual execution and function call path. Permutating all possibilities is intractable. Therefore, a heuristic algorithm is essential.

At compile time, a static analysis module scans source code, identifies unsafe operations, and calculates unsafe pointer casting closure. A complete, but minimized closure would insert less primitives into the transformed source code. On the other hand, at run-time, a dynamic check and update module is activated through those inserted primitives to capture the affected events. At each migration/checkpointing point the precise thread state can be constructed, or a warning message is issued to users to abort any un-recoverable unsafe operations, such as illegal library calls. With this pointer inference algorithm, *MigThread* becomes practical and applicable for more real applications.

The remainder of this paper is organized as follows: Section 2 introduces the thread migration/checkpointing package *MigThread*. Section 3 discusses pointer operations in C. In Section 4, we describe the pointer inference algorithm in detail. Some experimental results and microbenchmarks are shown in Section 5. Section 6 gives an overview of related work. Our conclusions and continuing work are presented in Section 7.

2   Thread Migration/Checkpointing

Migration/checkpointing schemes can be classified by the levels at which they are applied. They fall into three categories: kernel-level, user-level and application-level [1]. Although only application-level ones support portability across heterogeneous platforms, they suffer from transparency and complexity drawbacks.

2.1   *MigThread*

MigThread is an application-level thread migration/checkpointing package [5], designed to take advantage of the portability and scalability in heterogeneous distributed computing environments. *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 thread state efficiently. Its power can improve the "weak transparency" drawback of application-level schemes. The thread state is moved from its original location (libraries or

kernels) and abstracted up to the language level. Therefore, the physical thread state is transformed into a logical form to achieve platform-independence. All related information with regards to stack variables, function parameters, program counters, and dynamically allocated memory regions, is collected into certain pre-defined data structures. *MigThread* also supports user-level memory management. Therefore, both thread stacks and heaps are moved out to the user space and handled by *MigThread* directly. Since the address space of a thread could be different on source and destination machines, pointer values may be invalid after migration/checkpointing. It is the preprocessor's responsibility to identify and mark pointers at the language level so that they can easily be traced and updated later.

The run-time support module constructs, transfers, and restores thread state dynamically. Since the preprocessor has collected enough information, thread state is constructed promptly [5, 6]. Internally, thread stacks and heaps are rebuilt, while pointers are updated. In addition, *MigThread* takes care of all heterogeneity issues, such as byte ordering, data alignment, and data/pointer resizing.

2.2   Migration/Checkpointing-Unsafe Factors

Migration/checkpointing-safety concerns precise state construction for successful migration/checkpointing. Since the thread state consists of thread stacks and heaps, heterogeneous migration/checkpointing schemes need to interpret all memory segments precisely. They usually have difficulties in dealing with programs written in type-unsafe languages, such as C [3], 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/checkpointing. Most schemes rely on type-safe programming styles [3, 4]. 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/checkpointing. 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.

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/checkpointing schemes will fail to determine their existence. *MigThread* issues compile time warnings of the potential risks.

Table 1. Assignments in Normal Form

Assignment	Explanation
$x = y$	Direct Copy Assignment
$x = \&y$	Direct Address-of Assignment
$x = *y$	Direct Load Assignment
$*x = y$	Indirect Copy Assignment
$*x = \&y$	Indirect Address-of Assignment
$*x = *y$	Indirect Load Assignment

3 Pointer Operations in C

From the programming language's point of view, only the unsafe pointer casting is of concern for migration/checkpointing-safety. *MigThread* attempts to trace all pointer casting cases and detect potential pointers in any memory location.

3.1 Data Updating Operations

When variables are initially declared, they refer to memory locations directly or indirectly. Later on, their memory contents can be modified to hold data with other types. According to the C syntax, memory contents can be changed by increment and decrement statements, assignment statements, and library calls. Since the increment and decrement statements only change the data values instead of the types, type casting can only come from the other two manners.

Assignment statements propagate the type obligation for the left-hand side to the right-hand side [7]. Combined with type casting, they are the major mechanism to hide pointers in non-pointer type memory locations. To simplify the presentation of the algorithm, normalized assignment statements are listed in Table 1. *MigThread* scans a complicated assignment and breaks it into several basic items without introducing actual temporary variables as in other analyses [2, 7]. Table 1 only illustrates the top-level address-of operator "&" and pointer dereference operator "*." Both **x** and **y** may contain more of such operators inside, and type casting qualifier "(τ_{opt})" might be inserted ahead. Without type casting, each memory location will use its default type declared within the program. Otherwise, pointers can be placed in non-pointer type locations by a special pointer casting.

Another way to change memory contents is using library calls, such as **memcpy()**, which directly copy memory blocks from one place to another in memory space. When they contain pointer fields, unsafe casting should be screened out.

3.2 Pointer Casting

Data in memory can be referenced by the following:

- *Named Locations*: Variables in stacks referring to unique memory locations
- *Unnamed Locations*: Fields within dynamically allocated memory blocks in thread heaps do not have static names

– *Pointer Dereference*: For pointer p, *p indicates the content of the memory location that p is pointing to

Pointer casting can be done directly or indirectly, and explicitly or implicitly. For the normalized assignment statements in Table 1, if pointers are sent into *named locations*, we say it happens *directly*. Otherwise, through *unnamed locations* or *pointer dereference*, the casting is defined as *indirectly*. The direct casting of non-scalar types like structures is not permitted in ANSI C [8]. However, we can always achieve this effect indirectly by *pointer dereferencing*. Structures are flattened down into groups of scalar type members which are cast one-by-one. Pointer casting might take place here if some members are pointers. For such cases, we say it happens *implicitly*. Otherwise it happens *explicitly*.

Sometimes type casting might happen in locations without valid static types. They can be created by **malloc()** without a real type association. Another case might be the misalignment in data copy. Some addresses might not point to the beginning of scalar type data after certain pointer arithmetic operations. Such "*notype*" cases should be recorded for future interpretation.

4 Pointer Inference System

Compile-time analysis techniques, such as point-to and alias analyses [9], can be classified by flow- and context-sensitivity. Flow-sensitive analysis takes into account the order in which statements are executed whereas context-sensitive analysis takes into account the fact that a function must return to the site of the most recent call, and produce an analysis result for each calling context of each function [2]. Flow and context sensitive approaches generally provide more precise results, but can also be more costly in terms of time and/or space.

In this paper, the proposed pointer inference system for hidden/unhidden pointer detection 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. We argue that it is 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 : We do not distinguish contexts to simplify our algorithm. For larger programs the exponential cost of context-sensitive algorithms quickly becomes prohibitive.

4.1 Pointer Inference Rules

After *MigThread* identifies the types of the left-hand and right-hand sides of assignments, the pointer inference system has to apply pointer inference rules, shown in Fig. 1 and Fig. 2, to detect unsafe pointer casting. The function $typeof()$ is used to check the data types in variables or memory locations. The

```
                    x = y ∈ Prog
                 typeof(x)!= POINTER
                 ( (typeof(y)==POINTER)
      ||((typeof(y)!=POINTER)&&(y∈Closure)) )
      ─────────────────────────────────────────
                 Pointer_Casting(x, y)


                    x = &y ∈ Prog
                 typeof(x)!= POINTER
      ─────────────────────────────────────────
                 Pointer_Casting(x, &y)


                    x = *y ∈ Prog
                 typeof(x)== SCALAR
                 ( (typeof(*y)==POINTER)
      ||((typeof(*y)!=POINTER)&&(*y∈Closure)) )
      ─────────────────────────────────────────
                 Pointer_Casting(x, *y)


                    x = *y ∈ Prog
                 typeof(x)== STRUCTURE
                 typeof(x.α)== SCALAR
              ( (typeof((*y).β)== POINTER)
            || ( (typeof((*y).β)!= POINTER)
               && ((*y).β ∈Closure) ) )
            offset(x.α)== offset((*y).β)
      ─────────────────────────────────────────
            Pointer_Casting(x.α, (*y).β)
```
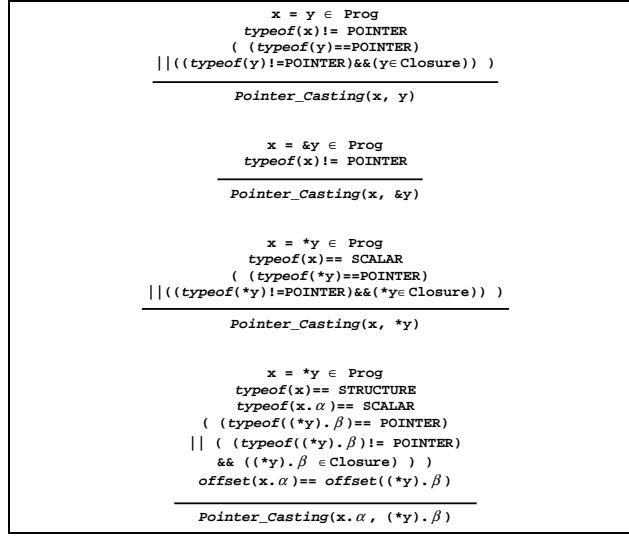
Fig. 1. Pointer inference rules for direct assignments

useful types are POINTER, SCALAR and STRUCTURE. The term *Closure* concerns the Pointer Casting (PC) Closure which is an approximate superset of possible variables and locations holding hidden pointers, maintained by *Pointer_Casting*() dynamically. Structure types are flattened down into scalar type groups whereas Union types' activation fields are traced since it is possible to create a pointer value under one field and then manipulate its value through another overlaid non-pointer field of the union. Letters α and β indicate the scalar fields which should be enclosed in PC Closure instead of entire structures or unions.

To simplify the presentation, we do not list type casting qualifiers in pointer inference rules, although in fact they are handled properly. At run-time, type casting phases modify dynamic types which are not recorded in the type system, and the dynamic type of the left-hand side determines the size of data to be copied from the location pointed to by the right-hand side of the assignment.

4.2 Static Analysis and Dynamic Check

With the intra-procedural algorithm, our pointer inference system focuses on functions themselves and ignores their inter-relationship for efficiency. Since it is not guaranteed to be able to predict the actual execution path, flow- and context-insensitive approach is adopted to avoid permutation of all possible cases.

The static analysis at compile time is the actual process that applies the pointer inference rules to the source code. Pointer Casting (PC) Closure is a conservative superset of hidden pointer locations, which is implemented in a red-black tree and used as a clue to insert run-time check primitives. After local variables and types are identified, the static analysis uncovers questionable
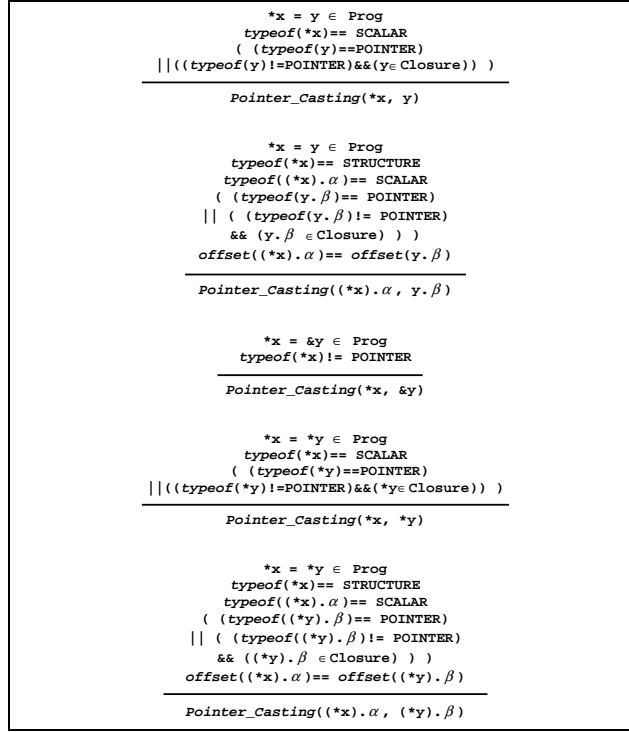
```
                    *x = y ∈ Prog
                  typeof(*x)== SCALAR
                  ( (typeof(y)==POINTER)
        ||((typeof(y)!=POINTER)&&(y∈Closure)) )
        ─────────────────────────────────────────
                  Pointer_Casting(*x, y)


                    *x = y ∈ Prog
                  typeof(*x)== STRUCTURE
                  typeof((*x).α)== SCALAR
                  ( (typeof(y.β)== POINTER)
                  || ( (typeof(y.β)!= POINTER)
                     && (y.β ∈Closure) ) )
                  offset((*x).α)== offset(y.β)
        ─────────────────────────────────────────
              Pointer_Casting((*x).α, y.β)


                    *x = &y ∈ Prog
                  typeof(*x)!= POINTER
        ─────────────────────────────────────────
                  Pointer_Casting(*x, &y)


                    *x = *y ∈ Prog
                  typeof(*x)== SCALAR
                  ( (typeof(*y)==POINTER)
        ||((typeof(*y)!=POINTER)&&(*y∈Closure)) )
        ─────────────────────────────────────────
                  Pointer_Casting(*x, *y)


                    *x = *y ∈ Prog
                  typeof(*x)== STRUCTURE
                  typeof((*x).α)== SCALAR
                  ( (typeof((*y).β)== POINTER)
                  || ( (typeof((*y).β)!= POINTER)
                     && ((*y).β ∈Closure) ) )
                  offset((*x).α)== offset((*y).β)
        ─────────────────────────────────────────
              Pointer_Casting((*x).α, (*y).β)
```

Fig. 2. Pointer inference rules for indirect assignments

assignments by their components on the left-hand side (LHS) and right-hand side (RHS), and suspicious library calls by their parameters. These items could be global variables, local variables in stacks, global/local memory locations in heaps, and function/library calls. They have to be extracted out and applied against the pointer inference rules. If LHS is not in PC Closure and unsafe pointer casting happens in RHS, LHS will be inserted into PC Closure. If the generated PC Closure is not empty, a primitive is appended after each **memcpy()** and structure casting case to detect possible implicit pointer casting.

At run-time, the dynamic check section is activated by the primitives inserted at compile time. Another red-black tree is maintained to record actual unsafe pointer casting variables and locations, from which *MigThread* can decide whether it is safe to conduct migration/checkpointing. This pointer inference system contains both detection and recovery functionalities. Another option is to only maintain a simple detection strategy for minimal overhead by only determining whether unsafe pointer casting has occurred. PC Closure is not created by enumerating all possibilities as above. If pointer casting occurs, migration/checkpointing will abort even though its effect is removed later. This option is useful for large programs since a recovery system could be quite complicated. Users can choose this option in *MigThread*.
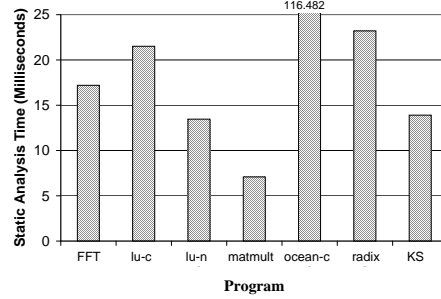
Fig. 3. Execution time of Pointer Inference Algorithm

4.3 Complexity

The space complexity of our pointer inference is $O(N_{prog})$, where N_{prog} is the size of the input programs. The major time complexity is in calculating the PC Closure, which is maintained on a red-black tree with maximum number of related assignment nodes N_{asn}, including large scalar and structure type assignments since these types are large enough to contain pointers. If the execution order of these assignments is the same as the order in the program, the best case will be achieved: $O(N_{asn} * logN_{asn})$. But if these two orders are completely opposite (one is the reverse of the other), we will get the worst case: $O(N_{asn}^2 * logN_{asn})$. To avoid it, we establish a threshold: after parsing repeats a certain number of times, we just append corresponding primitives after each related assignment. That means, instead of calculating the smallest PC Closure, we just take a bigger one. This will leave the extra overhead to the dynamic check whose complexity is simply $O(N_{asn} * logN_{asn})$. Therefore, major overhead can be avoided.

5 Experimental Results and Microbenchmarks

Our testing platform is an Intel Pentium 4 with 1.7GHz CPU, 384MB of RAM, and 700MB of swap space. To evaluate the overall pointer inference algorithm, we apply it to some programs, such as FFT, lu-c, lu-n, ocean-c and radix from the SPLASH-2 suite, matrix multiplication, and KS from the Pointer-Intensive Benchmark Suite [10]. Among them, most static analysis overheads are around 15 milliseconds, except the one for large ocean-c application which consumes 116 milliseconds, as shown in Fig. 3. Since this one-time cost is at compile time and is still relatively small compared to the application's execution time, the overhead of static analysis overhead is minimal.

The detailed information that migration/checkpointing systems can acquire is shown in Table 2 and Fig. 4. The term *shorter* indicates smaller sized data types, such as **char** and **integer**, which normally are not big enough to hold pointer values, and the term *longer* represents larger data types, such as **long** and **double**, which can encompass pointers. The terms *struct* and *pointer* indicate the structure and pointer types in C. The *dir* and *ind* concern values that can be copied into left-hand side directly and indirectly, respectively. The *notype*

Table 2. Assignment statement analysis test (unsafe cases in parentheses)

| Program	Line No.	Total Asn.	shorter		longer		struct		pointer		no-type	mem-cpy
			dir	ind	dir	ind	dir	ind	dir	ind		
FFT	1058	183 (0)	113 (0)	3 (0)	68 (0)	1 (0)	0 (0)	0 (0)	4 (0)	0 (0)	0 (0)	0 (0)
lu-c	1051	227 (0)	154 (0)	0 (0)	58 (0)	1 (0)	0 (0)	0 (0)	14 (0)	0 (0)	0 (0)	0 (0)
lu-n	829	132 (0)	64 (0)	0 (0)	58 (0)	1 (0)	0 (0)	0 (0)	9 (0)	0 (0)	0 (0)	0 (0)
matmult	418	43 (0)	30 (0)	0 (0)	6 (0)	1 (0)	0 (0)	0 (0)	6 (0)	0 (0)	0 (0)	0 (0)
ocean-c	5255	1946 (0)	917 (0)	0 (0)	613 (0)	3 (0)	0 (0)	0 (0)	410 (0)	3 (0)	0 (0)	0 (0)
radix	1017	218 (0)	152 (0)	0 (0)	44 (0)	1 (0)	0 (0)	0 (0)	21 (0)	1 (0)	0 (0)	0 (0)
KS	755	169 (0)	29 (0)	0 (0)	15 (0)	7 (0)	4 (0)	0 (0)	84 (0)	30 (0)	0 (0)	0 (0)
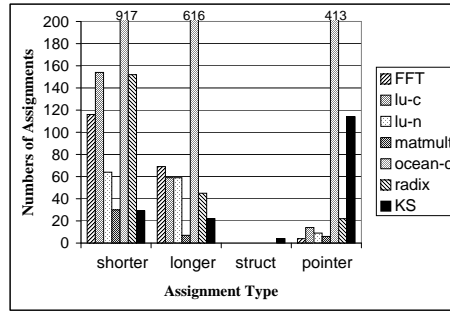
Fig. 4. Major features of benchmark programs

counts other random type assignments, and the *memcpy* shows the number of **memcpy()** system calls. The numbers in parentheses indicate the number of unsafe pointer casting instances. In this experiment, the *shorter* types are used slightly more than *longer* types. Both of them are used much more frequent than *pointer* types, which is the dominant factor in KS. Finally, structure casting rarely takes place.

Fortunately, among these benchmark programs, no unsafe pointer casting occurs. One major reason is that these programs come from well-known benchmark suites which are primarily created by professionals and type safety rules in C are followed restrictively. But in practice, we cannot count on programmers to achieve this. The pointer inference algorithm has to be applied to deal with the worst case so that programmers can work in any coding style without affecting migration/checkpointing functionality.

To evaluate the dynamic check at run-time, a microbenchmark program is used to investigate the maintenance of PC Closure which is the major operation to register/unregister variables and memory addresses. Fig. 5 illustrates different overheads when PC Closure is saved in a red-black tree with variant numbers of members. The operations with memory addresses normally take more time because they require extra time to convert addresses. Even so, their total cost is about 5-7 microseconds whereas the overhead for variables is just 2-4 microseconds. Since these run-time overhead values are small and increase very slowly as PC Closure expands, the cost of the dynamic check is negligible.
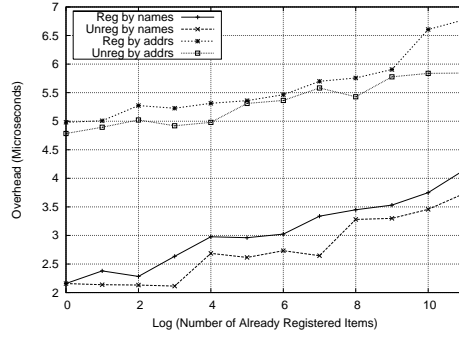
Fig. 5. Microbenchmark Results

6 Related Work

Among application-level migration/checkpointing schemes, the Tui system [3] discusses the safety issue, mainly from the point of view of type safety, without providing an effective mechanism about how to prevent it from happening or fixing it once it happens. Another package, SNOW [4], simply declares that it works only for "migration-safe" programs without further explanation or definition of "migration safety". Both of the above packages rely on programmers to avoid any unsafe cases during migration. Our previous work [6] identified some migration-unsafe features in C without providing a systematic solution.

Modern optimizing compilers and program comprehension tools adopt alias analyses to compute pairs of expressions (or access paths) that may be aliased, and points-to analyses to compute a store model using abstract locations [9, 2, 7]. Steensgaard presented a linear-time points-to analysis which contains storage shape graphs to include objects with internal structure and deals with the arbitrary use of pointers [9]. Andersen defined a points-to analysis in terms of constraints and constraint solving [11]. Yong, *et al.*, developed an "offset" approach to distinguishing fields of structures in pointer analysis and makes conservative approximations efficiently [8]. Wilson and Lam proposed a context-sensitive pointer analysis algorithm which is based on a low-level representation of memory locations to handle type casts and pointer arithmetic in C programs [12]. None of these can deal with pointer casting efficiently.

To deal with the unsafe type system in C, CCured [13] seeks to rule out all safety violations by combining type inference and run-time checking. Cyclone compiler performs a static analysis on source code, and inserts run-time checks into the compiled output at places where the analysis cannot determine that an operation is safe [14]. *MigThread* applies a lightweight pointer analysis to solve a different problem: detecting unsafe pointer casting effectively.

7 Conclusions

To overcome the restriction in existing migration/checkpointing schemes where programmers have to write type-safe programs, we have presented an intra-

procedural, flow-insensitive, and context-insensitive pointer inference algorithm, a lightweight pointer analysis, which conducts static analysis to predict possible harmful dynamic types and inserts run-time checks to detect hidden pointers so that the correct thread state can be constructed for migration/checkpointing. Experiments on real applications indicate that the overhead of this scheme is minimal. To reduce overhead, it can be configured to only detect the occurrence of unsafe pointer castings without tracing dynamic types although this might abort migration/checkpointing too conservatively.

This algorithm ensures correct migration/checkpointing whereas other schemes leave the uncertainty to application users. The following research is to provide a full-fledged support for third-party libraries whose procedural calls can introduce undetectable hidden pointers and memory blocks. Therefore, pointer inference algorithms can reach beyond ANSI C.

Acknowledgements

References

1. Milojicic, D., Douglis, F., Paindaveine, Y., Wheeler, R. and Zhou, S., Process Migration, ACM Computing Surveys, 32(8) (2000) 241-299
2. Rugina, R. and Rinard, M., Pointer Analysis for Multithreaded Programs, Proc. of the Conf. on Program Language Design and Implementation (1999) 77-90
3. Smith, P. and Hutchinson, N., Heterogeneous process migration: the TUI system, Tech rep 96-04, University of British Columbia (1996)
4. Chanchio, K. and Sun, X., Data Collection and Restoration for Heterogeneous Process Migration, Proc. of Int'l Parallel and Dist. Processing Symp. (2001) 51–51
5. Jiang H. and Chaudhary V., Compile/Run-time Support for Thread Migration, Proc. of Int'l Parallel and Distributed Processing Symp. (2002) 58-66
6. Jiang H. and Chaudhary V., On Improving Thread Migration: Safety & Performance, Proc. of the Int'l Conf. on High Performance Computing (2002) 474-484
7. Chandra, S. and Reps, T. W., Physical Type Checking for C, Proc. of Workshop on Program Analysis for Software Tools and Engineering (1999) 66-75
8. Yong, S. H., Horwitz, S. and Reps, T. W., Pointer Analysis for Programs with Structures and Casting, Proc. of the Conference on Programming Language Designand Implementation (1999) 91-103
9. Steensgaard, B., Points-to Analysis by Type Inference of Programs with Structures and Unions, Proc. of the Int'l Conf. on Compiler Construction (1996) 136-150
10. Pointer-Intensive Benchmark Suite, Computer Science Department, University of Wisconsin, Madison, http://www.cs.wisc.edu/ austin/ptr-dist.html
11. Andersen, L. O., Program Analysis and Specialization for the C Programming Language, PhD thesis, Dept. of Computer Science, University of Copenhagen (1994)
12. Wilson, R. P. and Lam, M. S., Efficient Context-Sensitive Pointer Analysis for C Programs, Proc. of the Conf. on Programming Language Design and Implementation (1995) 1-12
13. Necula, G. C., McPeak, S. and Weimer, W., CCured: type-safe retrofitting of legacy code, ACM Symp. on Principles of Programming Languages (2002) 128-139
14. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J. and Wang, Y., Cyclone: A safe dialect of C, USENIX Annual Technical Conference (2002) 275-288