Adaptation Point Analysis for Computation Migration/Checkpointing

Yanqing Ji
Elec and Comp Eng Dept
Wayne State University
48201 Michigan, USA
+01 3135770764
yqji@wayne.edu

Hai Jiang
Dept of Computer Science
Arkansas State University
72467 Arkansas, USA
+01 8706808164
hjiang@csm.astate.edu

Vipin Chaudhary
Institute for Scientific Computing
Wayne State University
48201 Michigan, USA
+01 3135775421
vipin@wayne.edu

ABSTRACT
Finding the appropriate location of adaptation points for computation migration/checkpointing is critical since the distance between two consecutive adaptation points determines the migration/checkpointing scheme's sensitivity and overheads. This paper proposes a heuristic adaptation point placement algorithm to improve the computation migration/checkpointing schemes' performance in terms of sensitivity and flexibility. This heuristic algorithm enables automatic and transparent insertion of checkpoints in user's source code.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Distributed applications.*

General Terms
Algorithms, Performance, Experimentation.

Keywords
Adaptation points, checkpointing, process migration, thread migration.

1. INTRODUCTION
Thread/process migration and checkpointing can be used to achieve load balancing, idle cycle utilization, resource sharing, and fault tolerance [1][2]. To enable these schemes to work in heterogeneous environments, we have developed *MigThread* to abstract computation states at the language level for portability [2]. However, one needs to determine how to insert adaptation points for potential migration or checkpointing.

An adaptation point is a location in a program where a thread/process can be correctly migrated or checkpointed. Since the overheads associated with constructing, transferring, and retrieving computation states are not negligible [3], finding appropriate adaptation points is essential. The distance between two consecutive adaptation points determines the migration algorithm's sensitivity and overheads. If they are too far apart, the applications might be too insensitive to the dynamic situation. But if they are too close, the related overheads will slow down the actual computation. In this paper, we propose a heuristic adaptation point placement algorithm for application-level migration and checkpointing systems where the computation state can be constructed at the application level.

SAC'05, March 13-17, 2005, Santa Fe, New Mexico, USA.

2. HEURISTIC ALGORITHM FOR ADAPTATION POINTS
Since *MigThread* is implemented at application-level, we have to insert certain code into user programs in order to enable migration or checkpointing. But finding proper insert locations in the source code is hard. Instead of selecting actual adaptation points, our solution is to aggressively insert a lot of *potential adaptation points* at compile time, as shown in figure 1. At run time, a scheduler dynamically collects load information from all related machines, and once it determines that a thread/process needs to be migrated to another machine, it sends a signal to *MigThread*. The signal handler will set the migration flag (*mth_flag*), and the corresponding thread/process will be actually migrated or checkpointed at the next *potential adaptation point.*

```
for (i = 0; i < upperBound; i++) {
        sum += func(i);
        if (mth_flag == 1) {
                checkpoint();          potential_adaptation_point()
        }
}
printf("%d", sum);
```

Figure1. *Potential adaptation points* in a loop

To insert *potential adaptation points*, the preprocessor has to analyze the structure and different components of user's source code. Usually, programs consist of loops, common non-loop code blocks, function calls, and library calls. Besides, the preprocessor has to take care of some special statements such as return, exit, and so on. Next, we describe the heuristic rules for inserting *potential adaptation points* under different conditions.

2.1 Loops

Taking care of loops in user's source code is very important since they consume most of the execution time in a program. In general, we apply the following rules:

- One *potential adaptation point* is inserted right after the last statement in each single loop.
- In case of nested loop, *potential adaptation points* are inserted inside the innermost loop.
- Within the same outer loop, if there are multiple nested loops sitting in parallel, one *potential adaptation point* needs to be inserted in each of them.
- For systems using relaxed memory consistency models, *potential adaptation points* should be inserted with pseudo-barriers and barriers.

The key idea is that we insert at least one *potential adaptation point* for each loop since we might not know the loop bounds at compile time.

2.2 Non-Loop Code and Functions

We argue that the non-loop code is never too long. Therefore, we can ignore non-loop code when looking for the proper places for *potential adaptation points*. However, recursive functions and nested calls must be considered since they could consume a long period of time. In summary, the following rules are applied:

- Non-loop instructions including branch instructions will be ignored since they usually do not consume much time.
- For each subroutine, at least one *potential adaptation point* is inserted. If no loop exists, we insert the *potential adaptation point* at the end of that subroutine.
- To ensure at least one *potential adaptation point* for each execution path, we insert a *potential adaptation point* before any "break" or "return" statement.
- For systems using relaxed memory consistency models, *potential adaptation points* should be inserted with pseudo-barriers and barriers.

2.3 Library Call and I/O Operations

Without the source code of libraries, the preprocessor cannot insert *potential adaptation points* into the library functions. 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.

I/O operations also have the potential to bring long period of time between two consecutive *potential adaptation points*. The reason is that the cost of I/O operations depends on the data volume and external factors such as network bandwidth. The assumption is that I/O is not a proper time for migration/checkpointing.

3. EXPERIMENTAL RESULTS

To evaluate the overall overheads associated with our algorithm, Matrix Multiplication, Molecular Dynamics (MD) simulation and several applications from the SPLASH-2 application suite are chosen for experiments, as shown in Table 1. It shows that matrix multiplication is the only application whose overhead ratio is greater than 2% because it is a computation-intensive application with many small loops. For about half of the applications, their overheads are less than 1%. Therefore, the overhead introduced by our algorithm is acceptable.

Table 1. *Potential adaptation points* overhead

Program	Input size	Exec. time without adaptation points (us)	Exec. time with adaptation points (us)	Over head ratio (%)
FFT	1,024 pts	3,574	3,620	1.287
LU-c	512 x 512	2,270,464	2,293,040	0.994
LU-n	128 x 128	40,135	40,754	1.542
MatMult	128 x 128	146,810	149,883	2.093
RADIX	262,144 keys	918,865	921,939	0.334
MD	5,286 atoms	18,823,604	18,903,475	0.424

The interval between two consecutive *potential adaptation points* determines the scheme's response time. Thus, we also tested all the intervals in each application (not given because of the length limit). The results show that more than 99.99% intervals are less than 10 *us,* and none of them is greater than 10^4 *us*. Therefore, this experiment is consistent with our previous prediction: the non-loop code is never too long.

4. CONCLUSIONS

We have proposed a heuristic adaptation point placement algorithm which can be incorporated into application-level computation migration/checkpointing packages, such as *MigThread,* in heterogeneous environments. One of the major advantages of our scheme is that it is generic for both thread/process migration and checkpointing. Since it can tolerate more adaptation points, the applications can be more sensitive to their dynamic situations. Furthermore, our scheme has the potential to work with more complex schedulers, which could be very important for meta-computing or Grid computing.

5. REFERENCES

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

[2] H.Jiang and V. Chaudhary, "Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems. ", In *Proceedings of the 37th Hawaii International Conference on System Sciences*, Hawaii, USA, January 2004.

[3] H. Jiang, V. Chaudhary and J. P. Walters, "Data Conversion for Process/Thread Migration and Checkpointing", In *Proceedings of the International Conference on Parallel Processing* (ICPP), Kaohsiung, Taiwan, October 6-9, 200