Distributed Speculative Parallelization using Checkpoint Restart

Devarshi Ghoshal, Sreesudhan R Ramkumar, Arun Chauhan*

School of Informatics and Computing, Indiana University, 150 S. Woodlawn Ave., Bloomington, IN 47405, USA
{dghoshal,ramakris,achauhan}@indiana.edu

Abstract

Speculative software parallelism has gained renewed interest recently as a mechanism to leverage multiple cores on emerging architectures. Two major mechanisms have been used to implement speculation-based parallelism in software, software transactional memory and speculative threads. We propose a third mechanism based on checkpoint restart. With recent developments in checkpoint restart technology this has become an attractive alternative. The approach has the potential advantage of the conceptual simplicity of transactional memory and flexibility of speculative threads. Since many checkpoint restart systems work with large distributed memory programs, this provides an automatic way to perform distributed speculation over clusters. Additionally, since checkpoint restart systems are primarily designed for fault tolerance, using the same system for speculation could provide fault tolerance within speculative execution as well when it is embedded in large-scale applications where fault tolerance is desirable. In this paper we use a series of micro-benchmarks to study the relative performance of a speculative system based on the DMTCP checkpoint restart system and compare it against a thread level speculative system. We highlight the relative merits of each approach and draw some lessons that could be used to guide future developments in speculative systems.

Keywords: Speculative parallelization, clusters, checkpoint restart

1. Introduction

Speculation has been proposed as a mechanism to parallelize loops that cannot be parallelized statically due to dependencies that are impossible to resolve accurately at compile time [1]. In recent years speculation has gained heightened interest as a mechanism to leverage the multiple cores on current and emerging processors [2, 3, 4, 5, 6]. Strategies to implement speculation rely on sharing some process state between one or more speculative threads and one or more verifier threads. The results from the verifier threads serve to guarantee the correctness of speculative execution as well when it is embedded in large-scale applications where fault tolerance is desirable. In this paper we use a series of micro-benchmarks to study the relative performance of a speculative system based on the DMTCP checkpoint restart system and compare it against a thread level speculative system. We highlight the relative merits of each approach and draw some lessons that could be used to guide future developments in speculative systems.

*Corresponding author.

1877–0509 © 2011 Published by Elsevier Ltd. Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.044
An alternative approach to implementing speculation in software is to use transactional memory [7]. This approach provides a simpler programming model than thread-level speculation. However, software transactional memory continues to suffer in performance relative to thread level parallelism and is an active area of research.

We introduce a third approach to implementing software-based speculation, using checkpoint/restart (CR). CR is a technique that finds wide application in fault tolerant computing [8]. To the best of our knowledge it has not been applied to software-level speculative parallelization before. Even though CR has been known for several decades now, the push toward exascale computing has resulted in increased research activity on fault tolerance recently as hardware fault rates are expected to greatly increase with exascale machines. The result is a thrust toward efficient implementation of fault tolerance libraries and even incorporating fault tolerance directly within MPI [9, 10]. With advances in fault tolerance technology, and specifically CR, it has become possible to consider CR as a way to duplicate and unroll processes dynamically for speculative parallelization. CR provides a semantically cleaner mechanism for speculation than approaches that require intervention by the kernel. More importantly, they provide a path to distributed speculation over clusters, and using a hybrid model, have the potential to reach the performance afforded by shared memory speculative systems.

In this paper we use the CR library called DMTCP (Distributed MultiThreaded CheckPointing), from Northeastern University [11] to implement a system, that we call FastForward, which allows speculative parallelization not only within a single shared memory node, but across nodes in a distributed environment. Further, our implementation lets us leverage the high speed interconnect networks, if available, for process migration and exchanging data for the validation step. We present the design of our system and evaluate it by measuring the various overheads within FastForward through a sequence of micro-benchmarks. We have also implemented a source-level C compiler to simplify the task of specifying speculation in most common cases.

2. Design

FastForward has two main components, a simple application programming interface (API) that makes speculation available within a standard C program, and a run time system. FastForward is designed to operate at the user level, without requiring any kernel patches or modules, for maximum portability. This decision puts some constraints on the possible implementation strategies. For example, it precludes an implementation based on modifying operating system interrupt handlers.

2.1. The API

The first step in optimizing a program using FastForward is to identify sections in the code that will use speculation. Each such code section is enclosed within a conditional statement, as shown in Figure 1. Logically, the process is forked and one of the two processes is marked a verifier and the other a speculator. Normally, the speculator will finish earlier than the verifier, which executes the original safe version of the code. The speculator then creates a separate thread to validate the results against those produced by the verifier, and continues on. After the verifier finishes it also creates a validation thread that coordinates with the validation thread created by the speculator. Performing the validation in a separate thread ensures that both the speculator and the verifier can continue computations without waiting for the validation to finish, as long as sufficient number of cores are available. As soon as the validation finishes one of the speculator or the verifier processes will be terminated based on the outcome. If the validation succeeds then the verifier is terminated and the speculator is now designated to be safe. Otherwise, the speculator is terminated.
The actual validation may be performed on either end, i.e., either on the thread created by the verifier or the speculator, based on a configuration parameter passed to FF\_Fork. All the details of forking, validator thread creation, and data transfer are hidden inside the API. Notice that in order to gain any real performance over the original code there must be at least two regions of code that can be executed speculatively, or at least a speculative region followed by a non-speculative (safe) region. This lets the latency of the verification process be hidden by concurrently executing the verifier for first speculative region with the second speculative region or the non-speculated region.

The validation is performed by comparing results of the region that is computed speculatively. A simple method to automate the validation step is to compare only the set of live variables at the end of the speculated region. In our current implementation this can be done either through our source-level compiler or through a callback function written by the user, for complicated cases that compiler may not be able to handle. Using an explicit comparison callback function affords a high level of flexibility in implementing the comparison operation. Thus, recursive pointer-based data structures could be easily compared using the knowledge of the data structures. For example, if a dynamically allocated data structure was created within the code region that is being speculated on then a simple byte comparison for validation may be misleading since pointer addresses are unlikely to match across the verifier and the speculator. Similarly, the comparison operation could be made to tolerate small differences. If speculation involves changing the order of floating point operations the results might vary within an acceptable margin of error. This requires algorithmic knowledge that a compiler is unlikely to have. In such cases a special callback function could be specified by the user for particular data structures or variables.

2.2. The Runtime System

Figure 2 shows the architecture of the runtime system of FastForward. Two different strategies are used for intra-node and inter-node implementations. When speculating within a node new processes are created using standard fork system call with copy-on-write semantics, as shown in Figure 2(a). The fork system call turns out to have a low overhead. For inter-node, or distributed speculation, we use the DMTCP library to checkpoint the running process and then transfer the checkpointed image to the remote node to start the duplicate process. In effect, this implements a remote fork. FastForward makes use of a helper MPI application to implement the distributed speculation, as Figure 2(b) shows. The checkpoint data are transferred using NFS that operates over GigE on our system\(^1\). The MPI helper process listens on a pipe. As soon as it receives an indication that the checkpoint data is being written, it starts the process of contacting the directory server and requesting for an available node. Thus, three activities progress concurrently: (a) writing of the checkpoint; (b) execution of the next speculative region; and (c) protocol for requesting an available remote node for remote-fork. As soon as the checkpoint is ready, the MPI helper sends a message to its peer on an available node to restart the process, thus finishing the remote-fork. The same

---

\(^1\)DMTCP checkpoints to the filesystem, which is only sharable over GigE. In future, we plan to checkpoint to memory, thus bypassing NFS.
MPI processes are also used to communicate data for validation once the speculative region finishes (not shown in Figure 2(b)).

Using the MPI helper has two advantages. First, it lets FastForward use high speed interconnection networks that might be available on a high-speed cluster and leverage MPI optimizations for data transfer. Second, it solves a practical problem on batch allocated clusters by enabling controlled remote process creation through the MPI helper.

We have implemented both inter-node and intra-node versions of FastForward. We have also implemented intra-node versions using sockets as well as shared memory transports for data communication for comparison. Section 5 experimentally compares the different communication mechanisms that have been implemented.

2.3. Multi-level Speculation

An important aspect of FastForward is its support of multi-level speculation. We believe that the ability to continue speculating without waiting for the last verification results is critical in obtaining efficiency in speculative parallelism, as Section 4 explains. For this purpose, FastForward implements a protocol to keep track of which nodes (MPI ranks) are currently available for computing. It uses a directory-based approach, where node 0 serves as the directory that keeps track of the available nodes. Thus, FF_fork contacts node 0 to request an available node and transmits the checkpoint data directly to the available node. Since the destination node expects checkpoint data to arrive from the requester, this allows us to use MPI’s efficient two-way communication primitives.

Figure 3 illustrates the progression of the protocol with an example. Node 1 speculates twice. For each speculation it forks off a verifier to check the results against its speculated region. To find an idle node, the speculator node contacts the directory server, which responds with the number of the idle node. The directory service also sends a message to the idle node informing it of the node that will be sending it the checkpoint data. Notice that the creation of the checkpoint and other bookkeeping can often be completely overlapped, thus minimizing the bookkeeping overheads. The protocol is implemented by the helper process on each node that allows the main computation to proceed concurrently. The helper process will likely get scheduled on a separate core, if there is one available on the node. With a slight modification, the protocol can be adapted to allow for multiple processes on each node, which would be useful to exploit multiple cores on each node.

A node can be in one of three states: idle, speculating, or verifying. Figure 4 shows the state transition diagram. Additionally, the directory server maintains a record of which nodes are available. We have omitted the states of the directory server for the sake of clarity.

A more sophisticated hierarchical directory service is possible to implement in order to achieve greater scalability, which we leave for future work.

Note that at any point of time there is at most one node that is in speculating state. All other nodes are in either verifying or idle states. We do not distinguish between non-speculative computing and computing done for verifying speculative computation—a node is in the verifying state in both cases. The exact node performing speculation can change over the course of the application. The system does not allow more than a specified number of concurrent verifiers. The “throttling” is implemented simply by delaying the acknowledgement from the directory server until a node becomes available, effectively delaying the completion of FF_fork.

The comparison of results could be performed either on the speculator or the verifier. If the speculation is expected to succeed in most cases then it would be more useful to perform
the comparison on the verifier. Similarly, if the speculation is expected to fail in a majority of cases then the result comparison is better done on the speculator. However, the latter case is likely to be rare. Nevertheless, FastForward lets the result comparison be done on either, through a tunable parameter.

3. Implementation Status

We have implemented FastForward in C++ using DMTCP and MPI. FastForward allows a tunable number of concurrent verifiers, and lets the speculation migrate across nodes. FastForward currently implements a flat directory server. DMTCP checkpointing is invoked through the API, however the restart must happen using a command, via a helper script. The MPI helpers implement the protocol to manage the runtime. Figure 5 shows the pseudocode for the protocol.

We have also implemented a preliminary source-level compiler that combines live-variable analysis with information flow analysis to determine which variables need to be verified at the end of a potentially speculative region. The compiler also lets the speculative regions be specified more cleanly using #pragma directives. In a large number of cases—those that do not have indirect array references or pointer-base aliasing—the compiler can automatically generate code for comparing the outcome of speculator and the verifier. The compiler discards temporaries that might be used within speculative or non-speculative versions of the region, and compares only those values that would actually get used in later parts of the program. We also note that the compiler does not automatically generate speculative versions of code, which is outside the scope of this paper.

4. Analysis

In order to estimate an upper bound on the amount of performance improvement that we can expect on our system suppose that there are \( k \) regions of code that can be speculated upon for each non-speculative region. Suppose that each region of code takes time \( T \) to finish, and each speculative execution of that code is \( s \) times faster. Further, suppose that speculation succeeds with a probability \( p \) each time a speculative computation is performed. For one non-speculative and \( k \) speculative regions, the total running time of the original code is \( T(k + 1) \). For speculation-enabled computation, the running time is given by,

\[
T + pk\frac{T}{s} + (1 - p)kT
\]

ignoring the overheads of remote process creation and result verification. Thus, the maximum speedup, \( S \), of the system with speculation is given by:

\[
S = \frac{T(k + 1)}{T + pk\frac{T}{s} + (1 - p)kT} = \frac{k + 1}{k + 1 + pk(\frac{1}{s} - 1)}
\]

Equation 1 can be used to make some key observations.
1. If we let the probability of speculation succeeding be close to 1 then the speedup approaches \( \frac{k+1}{k+1} \). If the ratio \( \frac{k}{s} \) is sufficiently larger than 1 then the speedup can approach \( s \). This implies that even when speculation is almost always successful, to get close to the ideal speedup of \( s \) the application must have sufficiently large number of speculative regions for each non-speculative code region. This count is for dynamic instances of code regions. Thus, a high \( \frac{k}{s} \) ratio might be achievable in a case where speculation is performed inside a loop.

2. If we let the speculation speedup \( s \) get close to \( \infty \) then Equation 1 reduces to Amdahl’s Law, giving an upper limit on the overall speedup of \( \frac{k+1}{k+1} \). With perfect speculation, the upper bound is \( k+1 \). This is a strict upper bound—for an application that has \( k \) speculative regions per non-speculative region, it is impossible to get a speedup of greater than \( k+1 \).

3. The parameter \( k \) can be interpreted as the depth of speculation. In other words, the speculative computation may proceed with further speculation, up to \( k \) times, without waiting for the verifier corresponding to the previous speculation to finish. The above equation shows that in order to get reasonable performance improvements in practice, it is important that the system support deep speculation, i.e., large values of \( k \). Clearly, the depth of speculation depends on the application, however, the underlying system also needs to be ready to support what the application demands.

4. Speculation incurs an efficiency cost. In order to support a speculation depth of \( k \) we need \( k+1 \) nodes as discussed in Section 2. With \( k+1 \) nodes the parallel efficiency is \( 1/(k+1+pk(\frac{1}{s})-1) \). When speculation has high probability of being correct \( (p \approx 1) \) the efficiency is close to \( 1/(1+k/k) \). Unfortunately, if we wish a high \( k/s \) ratio to achieve a speedup close to ideal, as discussed above, then we pay a price in lowered parallel efficiency that approaches \( s/k \).

The above abstract analysis is applicable to any speculative system. The accuracy of speculation and the speedup of speculative region over non-speculative region depend on the specific application and the algorithm employed to achieve speculation. The critical parameter that depends on the speculative system, rather than the application, is the depth of speculation it affords. FastForward can support arbitrary depths of speculation. More importantly, it makes deep speculation worthwhile by leveraging multiple nodes across a cluster. This enables an application to scale beyond cores on a single node.

As we illustrate through a careful set of experiments in Section 5, by overlapping communication and computation, FastForward is able to keep the overheads small. At the same time intra-node overheads are minimized by using a thread-based mechanism with shared memory, whenever possible.

5. Experimental Evaluation

We conducted a series of experiments to measure the overheads of various communication mechanisms that FastForward uses and supports.

We wrote a benchmark, `infreq-spec`, which performs CPU-intensive computation (we used matrix-multiply, but it could have been anything else) in three phases. We assume that these computationally intensive regions may be speculatively optimized, which was simulated by parallelizing the region. We ran our tests on a 3 GHz Intel Core 2 Duo machine with a total of 8 cores and 8 GB RAM, running Gentoo Linux 2.6 kernel, with gcc 4.3 compiler. Between any two speculative regions there is a safe computation phase that is large enough to hide the latency for validation. Thus, this example highlights the case where speculative regions are relatively infrequent such that there is no waiting period for data validation. Figure 6 shows the structure of the benchmark. As long as there are sufficient number of cores, or nodes, the overhead of data validation can be maximally hidden.

Figure 7 summarizes the results of running `infreq-spec` on increasing input sizes. Lines marked “original” show the running time of the original code without any speculation. Lines marked “Spec (succ)” denote runs with speculation enabled where speculation always succeeds. Finally, lines marked “Spec (fail)” are the cases when speculation is
Figure 7: Intra-node FastForward using different communication channels.

Figure 8: Running time of the benchmark under different versions of FastForward and the breakup of the running time.

enabled, but it always fails to produce correct results. Thus, these plots serve to bound the performance of our system in the best and worst case scenarios for most favorable application scenarios.

Note that the price paid for the overheads can be fine-tuned in two different ways:

- If the speculation is expected to succeed in most cases then we can assume that the speculative execution is on the critical path and offload validation on the verifier core, or vice-versa.
- With the distributed speculation capability, result validation can be offloaded onto a remote node, if it becomes a bottleneck.

As expected, speculation that uses shared memory for the validation step runs with almost no speculative overhead if the speculation fails. Otherwise, the running time remains practically unchanged. This clearly indicates that shared memory should be used for validating results, whenever it is available.

Figure 8 tabulates the running time and its breakup across different components for the infreq-spec benchmark on different versions of FastForward. “Data-transfer” time is the time to transfer the results for validation, “validation” is the time to compare values, and “FF_fork” is the total time it takes to do a FF_fork. We used a high performance cluster of dual-core Opteron processors connected by Infiniband as well GigE for this experiment. While there is a significant difference between speculating locally (on the same node) compared to remotely (another node), there is still a substantial improvement over non-speculative version.

In these experiments the amount of time spent on transferring data was a relatively small fraction of the total computation time. As a result, there is not much difference in the performance of distributed FastForward between using GigE relative to using Infiniband for transferring the data to be compared. This shows that for computationally intensive tasks, where the computation to communication ratio is high, any reasonable interconnection network can deliver acceptable performance.

The main performance difference of distributed speculation from intra-node speculation arises from the overhead of remote process creation, which is three orders of magnitude higher than a local fork. Clearly, even without considering the data transfer times for the validation step, local speculation should be preferred whenever possible. This
points to the value of a hybrid model that uses shared memory-based speculation locally and resorts to CR-based speculation only when necessary.

The tabular data is elaborated in the graph in Figure 8. Clearly, there is a gain in running the program with inter-node speculation. Somewhat surprisingly the cost of failed speculation is relatively small. We attribute this to the fact that separate threads were spawned for validating the data and there were extra cores available to run those threads so that the validation process did not interfere with the main computation. This corroborates our design decision to use separate threads for validation, especially for inter-node speculation. Thus, for infrequent speculation, especially, when cores are available and the verifier is on the critical path inter-node speculation provides a compelling option.

In order to estimate the behavior of inter-node FastForward on an application that might speculate frequently, we measured two metrics, process creation time and data transfer time. Figure 9 shows these two values for increasing process sizes and increasing data sizes. Process creation time is compared against the local fork (labeled “Intra-node (shared memory)”). Not surprisingly, there is multiple orders of magnitude difference as was also seen in the table in Figure 8. As our later experiments show, the bulk of remote process creation time goes in creating the checkpoint.

Data transfer times demonstrate that Infiniband is significantly faster than GigE, which justifies the overhead of using MPI helpers to aid rapid transfer of large amounts of data.

In order to test performance improvement trends, we devised a generic benchmark with the following tunable parameters:

\[ s = \text{speedup of the speculation over non-speculative region} \]
\[ p = \text{probability that the speculation produces correct results} \]
\[ d = \text{amount of data that needs to be compared to verify the speculative results} \]
\[ c = \text{size of the checkpoint} \]
\[ v = \text{maximum number of concurrent verifiers} \]

An important thing to note is that \( v \), the number of parallel verifiers, determines the maximum depth of speculation that the system affords. FastForward has no inherent limits on \( v \), although the number of available nodes on a cluster would limit \( v \) in practice.

Section 4 provided theoretical upper limits on performance improvements under different assumptions about some of these parameters. In order to measure the performance trends we conducted a series of tests by changing one parameter at a time. To isolate system overheads due to speculation the generic benchmark simulates the execution of back-to-back speculative regions.

Figure 10 summarizes the results of these tests, which were conducted on a 128-node cluster of dual-core 1 GHz AMD Opterons, running Linux 2.6, gcc 4.1.2, and OpenMPI 1.2.6. These tests show that the checkpoint size has a large impact on the overall performance, since it slows down the checkpointing as well as the data transfer process. Direct-memory checkpointing and incremental checkpointing could ameliorate this problem. On the other hand, increased probability of success has smaller than expected impact, especially for the initial portions. This could partly be due to the fact that the comparison for all these experiments was always done on the verifier. With low success probability of speculation the verifier becomes the bottleneck.
6. Related Work

Although the basic idea of FastTrack [5] is similar to FastForward, our implementation strategy differs significantly. While FastTrack uses page-level checks and requires a kernel module, FastForward is a completely user-level solution. Our approach does not depend on any profiling tool and provides a flexible approach to validating results. The throttling mechanism in FastTrack is a relatively orthogonal feature and could also be utilized in our FastForward system.

The aggressive compiler optimization and parallelization model used in several thread-level speculation methods [3] create hot regions of frequently executed code using compiler support. A previous execution history is maintained to keep track of the results and regions of frequently executed code. This model requires support from hardware, unlike FastForward. Another drawback of this model is that the hot speculative path is chosen based on execution history, expanding the application memory footprint. Finally, the model focuses primarily on control dependencies.

The method implemented in [12] defines a main thread that uses “copy or discard” mechanism to handle results which differs from our “progress / rewind and discard” mechanism. Since, the speculative parallel code will always be ahead of the non-speculative code, correct results would discard the safe version and move ahead with the unsafe version, creating another version of the safe code, resulting in reduced validation overhead.

Software Behavior Oriented Parallelization (BOP) [13] approach is similar to our approach, in principle. However, BOP incurs overheads of general protection and false-sharing. While our system might require an extra buffer copy to transfer validate data, it does not suffer from false-sharing. The data-checking mechanism in BOP segregates the program’s address space into disjoint groups implemented through a kernel-space mechanism. Our system is entirely in user space.

Several other recent efforts have been directed toward making speculation simpler by providing language-level support [14], improving the performance of validation step [15], comparing different approaches to thread-level speculation [16], or building speculative systems not described above [17, 18, 2, 3, 19, 4, 20, 6].

None of the speculative systems mentioned above provide cluster-based distributed speculation.

Figure 10: Performance trends with different parameters. The default values of the parameters are $s = 2$, $p = 1$, $d = 3.8$ MB, $c = 3.8$ MB, $v = 1$. 
7. Conclusion and Future Directions

In this paper we have presented and evaluated an implementation of the distributed speculation library system based on the DMTCP checkpoint/restart library. To the best of our knowledge this is the first time that a distributed speculation system has been implemented with this approach. Recent advances in CR technology has made this an attractive and feasible option. Our micro-benchmarks indicate that distributed speculation is not only feasible, but in fact produces reasonable speedups, especially in a hybrid model where local (intra-node) speculation uses a simpler more efficient system and the inter-node speculation uses CR. In cases where speculation could be frequently incorrect so that the verifier might be on a critical path, CR-based distributed speculation offers a compelling option since it can operate with minimal overheads and work well with multi-threaded applications.

Possible future directions include: (a) leveraging recently developed techniques for rapid comparison of dynamic data structures to compare recursive data structures [15]; (b) evaluating performance on multi-threaded programs; (c) exploring the possibility of implementing recently proposed techniques, such as [14], using a CR method; and (d) extending speculation support to large MPI programs.

8. Acknowledgments

We would like to thank the developers of DMTCP for developing and making the tool available. We especially thank Kapil Arya and Gene Cooperman for patiently answering all our questions. We thank Abhishek Kulkarni for helping with running our experiments on the Opteron cluster at Indiana University.

[1] L. Rauchwerger, D. Padua, The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization, in: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI), 1995, pp. 218–232.
[2] C. Zilles, G. Sohi, Master/slave parallelization, in: Proceedings of 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’02), 2002.
[3] L.-L. Chen, Y. Wu, Aggressive compiler optimization and parallelization with thread-level speculation, in: International Conference on Parallel Processing (ICPP), 2003.
[4] J. G. Steffan, C. Colohan, A. Zhai, T. C. Mowry, The STAMPede approach to thread-level speculation, ACM Transactions on Computer Systems (TOCS) 23 (3) (2005) 253–300.
[5] K. Kelsey, T. Bai, C. Ding, C. Zhang, Fast Track: A software system for speculative program optimization, in: Proceedings of the International Symposium on Code Generation and Optimization, 2009, pp. 157–168.
[6] M. F. Spear, K. Kelsey, T. Bai, L. Daless, M. L. Scott, C. Ding, P. Wu, Fastpath speculative parallelization, in: Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2009.
[7] N. Shavit, D. Touitou, Software transactional memory, in: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, 1995, pp. 204–213.
[8] D. P. Jasper, A discussion of checkpoint/restart, Software Age (1969) 9–14.
[9] P. H. Hargrove, J. C. Duell, Berkeley lab checkpoint/restart (BLCR) for Linux clusters, in: Proceedings of the Scientific Discovery through Advanced Computing (SciDAC), Journal of Physics: Conference Series, 2006, pp. 494–499.
[10] J. Hursey, J. M. Squyres, T. I. Mattox, A. Lumsdaine, The design and implementation of checkpoint/restart process fault tolerance for Open MPI, in: Proceedings of the IEEE International Parallel and Distributed Symposium (IPDPS), 2007.
[11] J. Ansel, K. Arya, G. Cooperman, DMTCP: Transparent checkpointing for cluster computations and the desktop, in: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2009.
[12] C. Tian, M. Feng, V. Nagarajan, R. Gupta, Speculative parallelization of sequential loops on multicore, International Journal of Parallel Programming 37 (5) (2009) 508–535.
[13] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, C. Zhang, Software behavior oriented parallelization, in: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2007, pp. 223–234.
[14] P. Prabhu, G. Ramalingam, K. Vaswani, Safe programmable speculative parallelism, in: Proceedings of the ACM SIGPLAN 2010 International Conference on Programming Language Design and Implementation, 2010.
[15] C. Tian, M. Feng, R. Gupta, Supporting speculative parallelization in the presence of dynamic data structures, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2010.
[16] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. C. Veleinbaum, C. D. Polychronopoulos, On the performance potential of different types of speculative thread-level parallelism, in: Proceedings of the 20th Annual International Conference on Supercomputing (ICS), 2006.
[17] M. Gupta, R. Nim, Techniques for speculative run-time parallelization of loops, in: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC), 1998.
[18] D. Brumine, S. Devabhaktuni, S. Amarasinghe, Softspec: Software-based speculative parallelism, in: Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization, 2000.
[19] M. Cintra, D. R. Llanos, Toward efficient and robust software speculative parallelization on multiprocessors, in: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2003, pp. 13–24.
[20] T. A. Johnson, R. Eigenmann, T. N. Vijaykumar, Speculative thread decomposition through empirical optimization, in: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2007, pp. 205–214.