Esodyp+: Prefetching in the Jackal Software DSM

Michael Klemm,[1] Jean Christophe Beyler,[2] Ronny T. Lampert,[1]
Michael Philippsen,[1] and Philippe Clauss[2]

[1] University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3
91058 Erlangen
Germany
{klemm, philippsen}@cs.fau.de
ronny.lampert@gmail.com

[2] Université de Strasbourg
LSIIT/ICPS
Pôle API, Bd Sébastian Brant
67400 Illkirch-Graffenstaden
France
{beyler, clauss}@icps.u-strasbg.fr

Abstract. Prefetching transfers a data item in advance from its storage location to its usage location so that communication is hidden and does not delay computation. We present a novel prefetching technique for object-based Distributed Shared Memory (DSM) systems and discuss its implementation. In contrast to page-based DSMs, an object-based DSM distributes data on the level of objects, rendering current prefetchers for page-based DSMs unsuitable due to more complex data streams. To predict future data accesses, our prefetcher uses a new predictor (Esodyp+) based on a modified Markov model that automatically adapts to program behavior. We compare our prefetching strategy with both a stride prefetcher and the prefetcher of the Delphi DSM system. For several benchmarks our prefetching strategy reduces the number of network messages by about 60 %. On 8 nodes, runtime is reduced by 15 % on average. Hence, network-bound programs benefit from our solution. In contrast to the other predictors, Esodyp+ achieves a prediction accuracy above 80 % with only 8 % of unused prefetches for the benchmarks.

1 Introduction

Today's high-performance computing landscape mainly consists of Symmetric Multi-Processors (SMPs) and clusters [1]. SMPs provide a hardware-managed global address space; clusters are assembled from nodes with private memory. Thus, programmers must explicitly communicate (e. g. with MPI's send/receive) to access remote data. Software-based Distributed Shared Memory systems (S-DSM), e. g. TreadMarks [8], Delphi [12], or Jackal [14], provide a shared-memory illusion on top of the distributed memory. The DSM adds a high-latency level (the remote memory) to the memory hierarchy of the nodes (registers, caches, main memory); performance drops due the growing data access latencies.

Prefetching provides a solution to this problem by prematurely requesting data before it is needed. Most current hardware (e. g. Intel Itanium, IBM POWER) offer prefetch instructions, that let the CPU asynchronously fetch data from main memory. Instead of adding such instructions at compile time, Esodyp [2] inserts prefetches into the program *while it is executed*; the inserted

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

prefetches are then executed by the CPU as usual. Esodyp continuously monitors all accesses to local memory in a Markov-like model, allowing it to predict future accesses. Esodyp+ extends Esodyp for use in an S-DSM on a cluster. It is implemented in Jackal, an object-based DSM, whose compiler prefixes each object access with an *access check* to test whether or not the object is cached on the local node. If not, the runtime system (RTS) requests the object from its home node. Esodyp+ extends the checks to monitor data access patterns in order to predict future accesses and prefetches the objects by bulk transfers.

Obviously, a dynamic prefetcher is superflous for embarrassingly/pleasingly parallel applications or if data access patterns can be analyzed and optimized statically. In contrast, network-bound applications with complex data access patterns benefit from our prefetching solution.

Section 2 covers related work. Section 3 introduces the Jackal DSM protocol. Section 4 presents the modified Markov model and the necessary extensions of the Esodyp+ approach for an S-DSM. Section 5 discusses issues of the Esodyp+ integration into the Jackal DSM. Section 6 shows the performance gain that Esodyp+ achieves and compares it to two known predictors.

2 Related Work

The related work can roughly be divided into a hardware axis and a size axis. With respect to hardware, prefetching occurs either (1) for single CPUs/SMPs or (2) in clusters, especially in S-DSMs. With respect to size, either (A) a single data item is prefetched or (B) a bulk transfer is used.

Almost all current CPUs provide prefetch instructions for single data items (1A). There are also projects, e. g. ADORE [10] or Chilimbi/Hirzel's work [4], that dynamically insert prefetches into programs. Modern hardware platforms with caches show a simple form of bulk prefetching (category 1B) since a whole cache line is "prefetched" upon memory accesses. On clusters, prefetching single data items (2A) is prohibitively expensive. As our proposal is in category 2B, we present this related work in more detail.

JIAJIA [9] uses a history of past accesses to predict. Delphi [12] predicts by means of a third order differential finite context method. Adaptive++ [3] relies on lists of past accesses to predict. In [6], an OpenMP DSM uses the inspector/executor pattern to determine future accesses. All these projects asynchronously request predicted pages one by one. For an object-based DSM this is not viable, as prefetching small objects cannot hide high network latencies.

Stride predictors [5] often separate data streams by using the instruction address as a hash key. While this works for stable access patterns/regular strides, it does not for the more complex access patterns of object-based DSM systems.

Whereas most prefetchers in S-DSMs request page by page, our approach bulk-transfers sets of objects. MPI programmers often use this technique to manually transfer large amounts of data by one MPI send/receive pair. Similarly, Java RMI [13] ships a deep copy of arguments to remote method invocations even if some of the shipped objects are not needed at the remote side. Jackal statically

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

```
1  int foo(SomeObject o) {          1  int foo(SomeObject o) {
2                                   2      if (!readable(o))
3                                   3          fetch(o, readable);
4      return o.field;              4      return o.field;
5  }                                5  }
```

Fig. 1. A function before (left) and after the insertion of an access check (right).

groups associated objects into larger ones [16] and, aggregating access checks, it requests a set of objects and ships it in bulk fashion [15]. Esodyp+ is not limited to *for* loops, but can be applied to arbitrary sequences of code, and it dynamically decides which objects to combine for transfer.

Instead of object combining, TreadMarks dynamically builds page groups, i. e. larger prefetching units [8]. A predictor continuously monitors page faults and decides which pages to group. The predictor is also capable of ungrouping pages if too much false sharing is caused. Our approach is similar in that it requests a set of objects during a prefetching request. However, the to-be-prefetched objects are not grouped and un-grouped, but the set is dynamically formed at each prediction step as necessary.

3 The Jackal DSM Protocol

Jackal [14] implements an object-based DSM for Java on clusters. Instead of cache lines or pages, data transfer and memory consistency are implemented on the granularity of Java objects to reduce false sharing. Jackal respects the Java Memory Model (JMM) [11] and provides a single system image to programmers.

Jackal inserts access checks into the Java code and compiles to a native executable, e. g. for IA-32 or Itanium. Fig. 1 shows a bit of original Java code (left) and the added check as Java-like pseudo-code (right). If the accessed object is not locally available, a request is sent to the *home-node* that stores the object's master copy. This request is answered with message that contains the requested data. Thus, the delay caused is roughly two times the network latency plus the cost of object serialization and deserialization. A prefetcher suffers from about the same latency for its request, but reduces the number of blocking waits by requesting the next accessed objects at the same time.

To uniquely identify each object that is allocated in the DSM, a Global Object Reference (GOR) is created. A GOR is a tuple *(alloc-node, alloc-address)*, where *alloc-node* is the logical rank of the node that created the object and *alloc-address* is the object's address on that node. The GOR is fixed during an object's life time; only if the object is reclaimed by the garbage collector, the GOR is released and may be recycled. When a non-local object is received, it is stored in a local caching heap and it is assigned a local address in the local address space for efficient object access. The DSM system maps these local addresses to their corresponding GORs and hence to the home-nodes (for status or data updates).

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

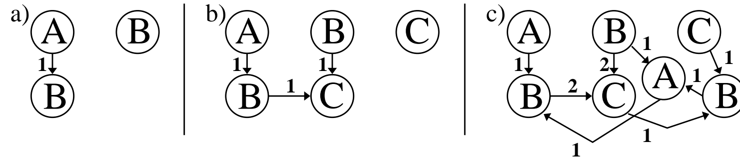

Fig. 2. The Esodyp+ graph (with a depth of 2) after two accesses (a), three accesses (b), and after the full sequence has been passed to the model (c).

Whereas Jackal mostly transfers individual objects, arrays are handled differently for performance reasons. Since transferring an array as a whole is not an option, the RTS partitions it into *regions* of 256 bytes. Thus, a failing access check requests only one region from the array's home-node.

4 Esodyp+

Esodyp+ extends Esodyp, the Entirely Software DYnamic data Prefetcher. Esodyp is a dynamic predictor that models address sequences with a variation of the classic Markov model [2]. This section briefly presents the model, how it reacts to the addresses passed by the DSM framework, and how it can help to prefetch objects. A complete explanation can be found in [2].

Classical Markov predictors [7] use two major parameters: *depth* and *prefetching distance*. The *depth* defines the number of past items used to calculate the predictions. The *prefetching distance* defines which item will be prefetched; a value of 1 means that we predict one next element. Esodyp+ is more flexible by creating and applying a graph instead of these parameters. With this graph, the model can define a maximum depth and can handle all smaller depths simultaneously. When predicting the next N accesses, Esodyp+ uses counters to prioritize the predictions. This is a major difference to other table-driven models [7].

To illustrate graph creation, let us assume Esodyp+ sees the following sequence of addresses: A, B, C, B, A, B, C. Fig. 2(a) shows the depth-2 graph after the first two accesses. For depth 2, the model takes into account the last two accesses. The arrow signifies that, after an access to A, an access to B occurred. For nodes without a successor, the predictor cannot predict anything. The single node B symbolizes that, if all we know is there was an access to B, nothing can be predicted. The edge label 1 indicates that Esodyp+ has seen the sequence (A, B) once. Graph construction evolves to Fig. 2(b) when the next address is seen. Two more nodes are added to the graph; C is attached to both Bs. This symbolizes that C occurs both after accessing B and after a sequence (A, B). After the whole sequence has been processed the full graph has 7 nodes, see Fig. 2(c). There, the edge label 2 indicates that an edge has been followed twice. By keeping in memory the current position in the graph, if the next access maps to a known pattern, we can predict in constant time. This is a major benefit compared to other predictors that perform calculations in order to predict.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

Moreover, Esodyp+ does not just handle simple addresses but it can store any type of information in its prediction model. Hence, we could directly map GORs to the model and would get an exact prediction. However, for each monitored GOR, at least one node in the graph would be created. Since this would make the model unmanageable for large working sets, it has to be kept compact without losing prediction accuracy. Our solution to this problem is similar to a stride prefetcher, as we compute the differences between two subsequent GORs and store only the difference in the model.

Originally, Esodyp was implemented using a *construction phase* to create the model that later is used in the *prediction phase*. Esodyp+ merges both phases and emits predictions even while it is constructing. This helps to reduce the overhead of the model by starting to predict earlier. This means that prediction strides must be recalculated if changes are made in the graph. Every time a change to the most probable child of a node is more recent than the last change of the current node, the prediction is recalculated. This makes Esodyp+ more dynamic and lightweight than other predictors. Like Esodyp, Esodyp+ triggers a flushing mechanism as soon as there are too many mispredictions [2].

5 Integration of a Predictor into Jackal

There are several ways a predictor for an object-based DSM must be special. Since predictors cannot guarantee accuracy, the DSM runtime must still check the validity of predicted GORs. Except for an object's creator, nodes do not know GORs without having accessed the object. Hence, only the home-node of an object can check a GOR for correctness. Thus, the predicted GORs are sent to the home-nodes and an object is only sent back for valid GORs; otherwise the request is safely ignored.

Predictors often only predict the next probable address. For each address a prefetch instruction is emitted into the instruction stream [4, 10]. Because of high network latencies in the DSM, prefetching of single objects do not give enough overlap of communication and computation. Hence, Esodyp+ emits predictions to the next N objects. If N objects are bulk-transferred, the program can continue without delay until the $(N+1)$th object is needed. In addition, the message count is reduced from $2N$ to 2 by serializing together N objects into a single message. While predicting the next object can be fairly accurate, the farther away a predicted access is, the smaller is the accuracy. This causes predictions of unused objects. In addition, a growing N increases the chance of false sharing, which in turn increases DSM protocol activity. Our measurements have shown that, on average, $N = 10$ is a good trade-off between false sharing and message reduction for the Jackal DSM.

Since prefetching alters the sequence of memory accesses (accesses that would regularly happen later in the execution are now performed earlier when a node prefetches data), it interferes with memory consistency. As a data item might be updated while a prefetch is outstanding, depending on the memory model, a prefetch either can continue or has to be canceled. Jackal implements the

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

JMM [11] even in the presence of a prefetcher. Simplified, each thread owns a private memory to cache its working set of data items that must be flushed when a synchronization point is reached. Hence, an update is usually not visible to other threads until all of them have reached a synchronization point as well and have flushed their caches. Prefetching blends well with such a weak memory consistency model. An active prefetch is not affected by concurrent updates that happen in the private cache of the updater. If the update hits a synchronization, it still does not affect the prefetch, since the JMM requires a synchronization for a cache refresh as well. As the requestor waits for the prefetch request to complete, it cannot reach a synchronization point in the meantime.

6 Performance

To evaluate the performance of Esodyp+ in Jackal, we compare it to both a stride predictor and the Delphi predictor on a Gigabit Ethernet cluster of Xeon 3.20 GHz nodes with 2 GB of memory and Linux (kernel 2.6.14.2). Since all prefetchers use the same RTS interfaces and the same maximum of N objects for each prefetching request, the measured differences are only caused by the overhead incurred and by the prediction quality. In all tests, each thread uses its own predictor model (since a global predictor does not make sense). Hence, the memory consumptions below are per thread and not globally.

We present four benchmarks and discuss the effects of the prefetchers. Since a predictor is useless in an embarrassingly parallel program, the benchmarks represent classes of applications that communicate with different access patterns. We feel that the selection is representative for applications that make use of general purpose DSMs. The results are the averages over 5 runs of each benchmark.

6.1 Predictors

To validate our predictor, we compared it two other predictors.

Our multi-stream **stride predictor** implementation [5] with a table of 128 bytes calculates a new stride as the difference between the current GOR and the last GOR. Using a *confidence counter*, predictions are only made if the same stride occurs a certain number of times in sequence. To give the stride prefetcher a better chance to keep recurring strides, we use a *dirty counter* that enables the predictor to ignore a different stride as long as the recurring stride reappears frequently enough in the sequence of strides. The next data accesses are predicted by adding the active stride to the current address. Stride predictors can only predict very regular accesses. As a GOR not only consists of a memory address but also contains a node rank, the Stride often mispredicts in non-array programs or for complex data distributions.

Our **Delphi predictor** implementation [12] continuously updates its information and uses a constant memory cache. As it cannot detect the frequency of sequences, rare sequences cause it to forget earlier sequences and to emit mispredictions. Delphi uses a hash function to map sequences to its table. In the best

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

Table 1. Runtimes and message counts for the benchmarks (best in bold).

	Nodes	Runtime (in seconds)				Messages (in thousands)			
		w/o	Stride	Delphi	Esodyp+	w/o	Stride	Delphi	Esodyp+
SOR	2	24.0	**23.7**	23.7	23.9	27.6	7.1	**5.4**	6.0
	4	13.2	12.3	**12.3**	12.4	83.1	22.7	17.9	**17.6**
	6	9.4	**8.6**	8.6	8.7	139.2	36.2	**29.9**	30.0
	8	8.0	**7.2**	7.3	7.3	196.0	53.9	**40.6**	42.3
Water	2	113.8	113.7	114.5	**102.3**	1593.0	1593.0	1593.0	**962.4**
	4	78.0	78.2	78.5	**62.1**	2651.6	2651.6	2651.5	**1593.9**
	6	64.4	64.4	64.5	**52.6**	3260.4	3260.5	3260.6	**1967.9**
	8	58.7	58.7	59.1	**49.6**	3756.5	3756.7	3758.4	**2248.9**
Blur2D	2	10.6	5.9	9.3	**3.6**	223.9	223.9	134.6	**33.4**
	4	6.9	6.6	5.9	**4.0**	385.4	385.4	190.0	**99.8**
	6	7.9	10.0	6.1	**5.0**	483.4	483.4	230.6	**166.2**
	8	8.8	13.5	7.1	**6.9**	581.5	581.5	277.2	**232.3**
Ray	2	**69.2**	71.7	71.8	69.9	9.2	5.9	8.7	**5.9**
	4	35.4	36.1	36.5	**35.2**	27.3	17.4	25.8	**17.1**
	6	24.1	24.2	24.6	**23.8**	45.4	29.0	42.1	**28.4**
	8	18.4	18.5	18.9	**18.3**	57.3	42.7	59.9	**41.1**

Table 2. Accuracy of different prefetchers and their unused but prefetched objects.

	Accuracy (in %)			Unused objects (in %)		
	Stride	Delphi	Esodyp+	Stride	Delphi	Esodyp+
SOR	81,09 %	**97,13 %**	95,39 %	2,48 %	**0,75 %**	6,61 %
Water	8,91 %	31,81 %	**67,81 %**	79,42 %	**7,88 %**	12,83 %
Blur2D	0,00 %	49,94 %	**76,81 %**	**0,00%**	1,96 %	10,36 %
Ray	79,99 %	0,00 %	**83,12 %**	6,31 %	100,00 %	**1,93 %**
Average	42,50 %	44,72 %	**80,78 %**	22,05 %	27,65 %	**7,93 %**

case, each sequence receives a unique index. However, the number of conflicts depends on the memory access pattern. Our implementation employs a 4096-entry hash table that uses the last three accesses as the hash key. Each entry contains a pointer to a structure containing a GOR and access check information. Hence, the total size of the table is 96 KB per thread. In a certain sense, Delphi is closest to a Markov model, as it uses a fixed depth of 3. But it cannot handle depths of 1 or 2 simultaneously. For n sequences $(A, B, *)$, Delphi needs n entries, each of which stores A, B, and the last element. Esodyp+, however, handles all the depths 1, 2, and 3, and stores the subsequence (A, B) only once. A prioritized linked list then covers the n possibilities.

6.2 Benchmarks

SOR iteratively solves discrete Laplace equations on a 2D grid (4,100×4,100; 50 iterations). It computes new values of a grid point as the average value of four neighboring points. Each thread of SOR receives a contiguous set of rows of the

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

grid. SOR only communicates at the boundaries of the partitions, at which the threads read the values of the grid points of other threads.

Although restructuring compilers for array-based languages may handle highly regular programs better, it is instructive how a prefetcher affects SOR. With a model size of 2.2 KB, Esodyp+ reduces SOR's runtime by about 9 % and the message count by about 78 %. As can be seen in Table 1, all prefetchers roughly achieve the same gain; Stride is slightly faster due to its lower internal overhead. Table 2 shows that Delphi is most accurate closely followed by Esodyp+. Stride does not outperform the others since the access sequence is not simple enough.

Water [17] simulates moving water molecules by means of an (N-square) N-body simulation. Work is divided by assigning molecules to different threads. After a thread has finished computing new directions and accelerations for its molecules in the current time step, it publishes these results for the other threads by means of a special class that implements both synchronization and simultaneous exchange of updates with other threads.

Water has a large working set (1,792 molecules) that is communicated after each time step, making it highly network-bound. The Stride cannot correctly predict, as the objects are scattered over the DSM. Delphi suffers from a number of conflicts in its database and from the high number of objects being accessed. In contrast, Esodyp+ builds an almost perfect model of 14.8 KB and reduces runtime by up to 20 % (for 4 nodes). The message count is decreased by about 40 %. No static analysis can achieve similar results for such irregular applications. Esodyp+ achieves 68 % accuracy (Table 2), which is twice as good as Delphi and about 8 times better than Stride. Stride predicts many unused objects since, once a decision is made, the program behavior has already changed.

Blur2D is a 2D convolution filter that softens a picture of 400×400 gray-scale points (20 iterations). It uses a 2D array of double precision values that describe the pixels' gray values. The value of a pixel is computed as the average of itself and its eight neighbors. For such a stencil computation (like SOR) accesses are difficult to predict because the parallelization does not fit to the data distribution of the DSM. While Jackal favors a *row-wise* distribution scheme, Blur2D uses a *column-wise* work distribution. Hence, false sharing and irregular access patterns make Blur2D highly network-bound.

In contrast to Stride and Delphi, Esodyp+ (5.4 KB model) saves roughly 20 % runtime on 8 nodes and the number of messages drops by 60 %. Esodyp+ predicts more accurately (almost 77 %) due to the additional information in its model. In contrast to Delphi and Stride, when constructing the request message, Esodyp+ first tries the most probable next access. If this leads to an object that has already been requested, it uses another edge of its graph to predict less probable but possible accesses. Hence, Esodyp+ prefetches not only the most likely sequence but also less probable sequences. For increasing node counts, it cannot compensate the false sharing. Stride's simplistic model is the reason for its poor behavior. Due to a high number of consecutive accesses to the same DSM region, Stride does not emit any good predictions. As of the confidence counter, Stride does not prefetch when a pattern change occurs. Disabling this

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

counter causes a 50 % slow-down due to mispredictions. Delphi's hash table is unable to provide enough slots to store the complex data access sequences of Water.

Ray renders a 3D scene constructed from 2,000 randomly placed spheres. The image is stored as a 2D array (500×500) of RGB values. For parallelization, the image is partitioned into independent sub-images that are assigned to the threads. Raytracing is inherently parallel: it has a read-only working set (the spheres) and a thread-local working set that is written (the sub-images).

On average, the prefetchers do not gain any performance, although Stride and Esodyp+ (1.5 KB model) save roughly 35 % of the messages. Delphi is unable to make correct predictions due to conflicts in the hash table. This leads to an accuracy of 0 % and an unused percentage of 100 %. Stride is able to keep a score of 79 % compared to the 83 % of Esodyp+. However, Stride makes more unused predictions (6,31 % vs. 1.93%). Transferring read-only data at initialization, the threads work on local data that is only transferred at the end of the computation. Obviously, for embarrassingly parallel applications prefetching does not improve performance. Variations in the cluster load cause the fluctuations in Table 1.

To **summarize**, let us compare the overheads of the predictors. When learning, Stride has the lowest overhead as it only considers strides relative to the last access. Delphi causes a higher overhead by computing the hash index for each access. Esodyp+ updates a few nodes and counters in the model, which also results in a higher overhead. When predicting, Delphi still calculates the hash index of the current access sequence, whereas Esodyp+ predicts by just following pointers to the most probable prefetch candidates. Hence, for stabilized models, Esodyp+ reaches the low overhead of a stride predictor. This is a major advandage of the Esodyp+ model, as it is able to modelize complex memory access behavior with a low prediction overhead once the model has been created.

7 Conclusions and Future Work

In this paper, we have shown that it is worthwhile to integrate a predictor into an object-based DSM, since generally predictors can compensate their overheads. Esodyp+, a novel Markov-based prefetcher performs better than two existing prefetchers. It is more precise and more efficient in predicting and emitting prefetches and reduces the message count by about 60 %. It reduces runtime by 15 % on an 8-node computation. On average, it achieves a 80 % accuracy compared to 45 % of the other predictors. Hence, our prefetcher is well-suited for network-bound applications with complex data access patterns.

Because of its Markov-like models, Esodyp+ automatically adapts to various access patterns and is only limited when a pattern is completely irregular. Therefore, instead of passing GORs, we work on adding structural data about object associations (i. e. connections between individual objects) to Esodyp+. We will also try to automatically choose an optimal prefetching distance at runtime, such that the distance best fits the program, reduces the potential of false-sharing, and speeds up the program. Another avenue of work is program phase detection.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --

By using Jackal's internal profilers, the predictor's impact on the program can be assessed. If prefetching is not efficient, it may be temporarily switched off to get rid of its runtime overhead.

References

1. TOP500 List. http://www.top500.org/, 2006.
2. J.C. Beyler and P. Clauss. ESODYP: An Entirely Software and Dynamic Data Prefetcher based on a Markov Model. In *Proc. 12th Workshop on Compilers for Parallel Computers*, pages 118–132, A Coruna, Spain, 2006.
3. R. Bianchini, R. Pinto, and C.L. Amorim. Data Prefetching for Software DSMs. In *Proc. Intl. Conf. on SC*, pages 385–392, Melbourne, Australia, 1998.
4. T.M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proc. ACM Conf. on PLDI*, pages 199–209, Berlin, Germany, 2002.
5. J.W.C. Fu, J.H. Patel, and B.L. Janssens. Stride directed prefetching in scalar processors. *SIGMICRO Newsletter*, 23(1-2):102–110, 1992.
6. W.-C. Jeun, Y.-S. Kee, and S. Ha. Improving Performance of OpenMP for SMP Clusters through Overlapping Page Migrations. In *Proc. Intl. Workshop on OpenMP*, Reims, France, 2006. CD-ROM.
7. D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.
8. P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. Winter 1994 USENIX Conf.*, pages 115–131, San Francisco, CA, 1994.
9. H. Liu and W. Hu. A Comparison of Two Strategies of Dynamic Data Prefetching in Software DSM. In *Proc. 15th Intl. Parallel & Distributed Processing Symp.*, pages 62–67, San Francisco, CA, 2001.
10. J. Lu, H. Chen, R. Fu, W. Hsu, B. Othmer, P. Yew, and D. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proc. 36th IEEE/ACM Intl. Symp. on Microarchitecture*, pages 180–190, San Diego, CA, 2003.
11. J. Manson, W. Pugh, and S.V. Adve. The Java Memory Model. In *Proc. 32nd ACM Symp. on PoPL*, pages 378–391, Long Beach, CA, 2005.
12. E. Speight and M. Burtscher. Delphi: Prediction-Based Page Prefetching to Improve the Performance of Shared Virtual Memory Systems. In *Proc. Intl. Conf. on PDPTA*, pages 49–55, Las Vegas, NV, 2002.
13. SUN Microsystems. RMI Specification, 1998. http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-title.doc.html.
14. R. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang, and H.E. Bal. Runtime Optimizations for a Java DSM Implementation. In *Proc. ACM-ISCOPE Conf. on Java Grande*, pages 153–162, Palo Alto, CA, 2001.
15. R. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang, C.J.H. Jacobs, and H.E. Bal. Source-Level Global Optimizations for Fine-Grain Distributed Shared Memory Systems. In *ACM Symp. on PPoPP*, pages 83–92, Snowbird, UT, 2001.
16. R. Veldema and M. Philippsen. Using Object Combining for Object Prefetching in DSM Systems. In *Proc. 11th Workshop on Compilers for Parallel Computers*, Seeon, Germany, 2004.
17. S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 24–36, St. Margherita Ligure, Italy, 1995.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT --