Automatic Prefetching with Binary Code Rewriting in Object-based DSMs

Jean Christophe Beyler,[1] Michael Klemm,[2]
Michael Philippsen,[2] and Philippe Clauss[1]

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

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

Abstract. Dynamic optimizers modify the binary code of programs at runtime by profiling and optimizing certain aspects of the execution. We present a completely software-based framework that dynamically optimizes programs for object-based Distributed Shared Memory (DSM) systems. In DSM systems, reducing the number of messages between nodes is crucial. Prefetching transfers data in advance from the storage node to the local node so that communication is minimized. Our framework uses a profiler and a dynamic binary rewriter that monitors the access behavior of the application and places prefetches where they are beneficial to speed up the application. In addition, we adapt the number of prefetches per request to best fit the application's behavior. Evaluation shows that the performance of our system is better than manual prefetching. The number of messages sent decreases by up to 89%. Performance gains of up to 73% can be observed on the benchmarks.

1 Introduction

The high-performance computing landscape is mainly shaped by clusters, which make up 81% of the world's fastest systems [1]. Clusters typically exhibit a distributed memory architecture, i. e., each node of the cluster has its own private memory that is not directly accessible by the other nodes. Software-based Distributed Shared Memory (S-DSM) systems strive to simulate a globally shared address space, alleviating the need to place explicit calls to a message passing library that handles communication (e. g. MPI [6]). Instead, a middleware layer accesses remote memory and ensures memory coherence. Examples of such S-DSM systems are JIAJIA [11], Delphi [14] or Jackal [16].

In addition to registers, L1/L2/L3 caches, and the nodes' local memory, the DSM adds another level (the remote memory) to the memory hierarchy. Remote memory accesses are much more expensive than local accesses, as the high-latency interconnection network has to be crossed. Hence, it is desirable to direct the application's memory accesses to the local memory as often as possible. For applications that do not offer such locality properties, performance drastically drops due to the high-latency remote accesses.

Prefetching provides a solution to this performance problem by requesting data from the storage location before it is actually needed. Most current hardware platforms (e. g. Intel Itanium, IBM POWER) offer prefetch instructions in their instruction sets. Such instructions cause the CPU's prefetching unit to asynchronously load data from main memory into the CPU. Executing the prefetch at the right time, data arrives just-in-time when needed by the program. Based on heuristics, compilers statically add prefetches to the code during compilation.

This paper presents a dynamic software system that automatically profiles and optimizes programs at runtime, outperforming manually optimized prefetching. During compilation, the Jackal DSM compiler [16] adds to the executable monitoring code that profiles memory accesses. A profiler classifies the memory accesses and enables prefetches if beneficial. Dynamic code rewriting keeps the the system's overhead low by interchanging the monitor code and the prefetching code in the executable. If prefetches are unprofitable, calls are completely removed, avoiding a slow-down of the application. However, if profitable, the code rewriter replaces the monitoring calls with prefetcher calls. The Esodyp+ prefetcher [10] is used as an example for our generic optimizer. It predicts memory access patterns and prefetches future memory accesses. While Esodyp+ requires manual placement of calls to its runtime functions, our system automatically inserts these calls if beneficial. Additional performance is gained by adapting the prefetching distance to the application's memory access behavior. For details on the implementation of the Esodyp+ predictor, refer to [10].

2 Implementation of Object-based DSMs

To understand the requirements for automatic prefetchers in an object-based S-DSM environment, this section first describes the basic DSM features and then explores the design space of DSM implementations.

Instead of using cache lines or memory pages for distributing data, object-based DSMs use objects for data transfers and for memory consistency. The DSM system checks for each object access if the accessed object is already available locally. If not, the DSM system requests the object from its storage location. The programming language's memory consistency model has to be respected, which involves invalidating or updating replicas of objects on other nodes. Testing for object availability can either be implemented in software or hardware.

In general, S-DSMs cannot use specialized hardware for the object access checks, as it should support commodity clusters. Some S-DSMs exploit the Memory Management Unit (MMU) of processors for access checks. The Operating System (OS) can use MMUs to protect memory pages against reading or writing. If a restricted page is accessed, the OS triggers the S-DSM and notifies it about the faulty access. After the S-DSM has loaded the page, the access is re-issued and the application continues. Delphi [14] and others [3, 8, 11] use this approach to implement a page-based DSM. However, memory protection can only be applied at the page level and, thus, renders this option unusable for object-based S-DSMs, as it causes false faults on local objects that reside on the same page.

```
 1         leaq -1308622848(%r8), %rdi
 2         shrq $5, %rdi
 3         movq %rdi,%rcx
 4         movq %fs:(0), %rdx
 5         movq thread_local_dsm_read_bitmap@TPOFF(%rdx), %rdx
 6         bt %rcx, (%rdx)
 7         jc .L28961
 8         movq %r8,%rdi
 9         call shm_start_read_object@PLT
10         movq %rax,%r8
11  .L28961:
```

Fig. 1. Example of an access check in assembly code, without prefetching.

Hence, for object-based S-DSMs, the access check has to be implemented in software, which is easy to do, because it often requires only a single bit test. Jackal [16], for example, relies on the compiler to prefix each object access with an access check that tests the object's availability on the local node.

We use the Jackal object-based DSM for our prototype implementation. Fig. 1 shows an assembler fragment of a read test as it is emitted by the Jackal compiler (prefetching is switched off). Lines 1–5 compute the object's read bit depending on the relative offset of the object in the heap; lines 6–11 test the bit and call the DSM runtime if the object is not locally available. As the runtime requests the object data from the object's home node and waits until data has arrived, each failing object access has a runtime penalty of roughly two times the latency of a network packet plus additional costs for object serialization and deserialization.

3 Memory Access Profiling and Dynamic Code Rewriting

Using a profiler, our automatic optimizer first needs to classify the access checks into categories to decide which require prefetching. Only a low overhead is acceptable for profiling, as the overhead must be compensated to reduce runtime. In addition, the optimizer adapts the number of prefetches per request message (the so-called *prefetching distance*) to optimally exploit prefetches depending on the application's access behavior. This section first explores the design space of such a dynamic optimizer and the profiler. It then covers the state machine of the profiler and discusses heuristics to adapt the prefetching distance.

3.1 Design Considerations

The main part of our dynamic optimizer is a low-overhead profiler. As prefetching is useless for access checks rarely executed or exhibiting random behavior, a classification by the profiler is crucial for the efficiency of the optimizer. There are several ways such a profiler can be implemented in the dynamic optimizer.

First, the monitoring code and the prefetching code could be accompanied by conditional guards. Implemented by a `switch` statement that decides between monitoring, prefetching, and no prefetching, the guards cause performance losses that result from additional instructions, increase loads on the memory bus, and

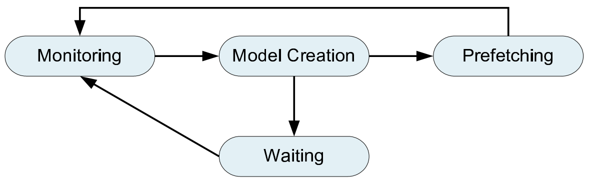

Fig. 2. State evolution of the access checks.

put a higher pressure on the branch prediction unit of the CPU. Hence, this option is expected to have a high overhead that is difficult to compensate.

With binary rewriting, the compiler prefixes access checks with monitoring code. After profiling, the code is either replaced with prefetching code or removed if prefetching is not beneficial. To remove code, the rewriter has to replace the code with `nop`s, as the subsequent binary code cannot replace the to-be-removed code fragment (moving code implies checking and modifying most branch instructions). However, `nop`s pollute the instruction cache and pipelines and, thus, cause undesired overheads. In addition, the profiler receives the program's local and remote accesses, as the code is placed in front of the access checks. Hence, the profile represents the general access behavior. Remote and local accesses are interleaved and distinguishing access checks that need prefetching and those that do not is impossible. An additional runtime overhead is caused, as the instrumentation code is always executed, even if only local objects are accessed.

Replacing the original access check code, however, provides low overhead, as only single calls must be changed to implement a state transition. It is possible to dynamically redirect the program's control flow without performance losses that would result from using `switch` statements or `nop`s. Furthermore, as a second advantage, access checks can be de-instrumented and replaced by their regular DSM counterparts. Thus, these access checks are executed without any overhead.

Directly integrating the monitor calls and prefetcher calls in access checks has two consequences. First, applications only incur an overhead in case of failing access checks. Therefore, if an application only accesses its local memory, there is no overhead since the prefetcher is never called. Second, as the predictor is only triggered during a failing access check, the created model only represents the application's behavior for accesses into the remote memory instead of modeling the memory references of every accessed object. Hence, the model only predicts the next remote accesses. If the model contained local references as well, they need to be filtered out, which causes an additional overhead when prefetching.

3.2 Profiling State Machine

We distinguish four states that characterize the behavior of an access check. Depending on the state, it may be beneficial to use prefetching (often executed, non-random access behavior) or not. Fig. 2 shows the state machine.

For access checks, the **Monitoring** state is the initial state. If an access check is often executed, it is sent to the *Model Creation* state. This is implemented with a counter per access check and a threshold t. In the **Model Creation** state, a mathematical model (e. g. a Markov model) determines the access behavior of an access check. If its behavior is unpredictable (i. e., random), prefetchers cannot correctly predict the next accesses and the access check state is changed to *Waiting*. If predictable, the access check proceeds to the *Prefetching* state and the prefetcher is enabled for this access check. **Waiting** represents the state in which the original access check code of the DSM is executed. To detect changes in the application's behavior so that the access check might later benefit from prefetching, the access check is periodically re-instrumented by the optimizer. To reduce the overhead and to avoid state thrashing, the time between re-instrumentations is increased at each cycle. **Prefetching** is enabled in the fourth state of an access check. As the access behavior of the access check is predictable, the prefetcher can predict the next accesses with high accuracy and it emits prefetch commands to prematurely request the data needed. If the prefetcher's prediction accuracy drops, the access check falls back to the *Monitoring* state for reassessment.

In the *Model Creation* state and the *Prefetching* state we use the Markov model predictor Esodyp+ [10], which uses past events to predict future events. It relies on the observation that events of the past are likely to repeat. Other Markov model based predictors could have been used as well [2, 9].

The profiler needs to efficiently identify access checks, as it profiles them separately. Since Jackal uses function calls for access checks (see Fig. 1), state transitions rewrite these function calls. We use the call's return address as a key into a hash table storing the profiler's data. In other DSM systems, the compiler could mark the access checks with identifying labels. During the *Initial* state, the hash table contains the hit counter for access checks. In the *Model Creation* state, the table stores a buffer collecting the requested memory addresses. Once the buffer overflows, the optimizer selects the most frequent access check, as it is likely to benefit most from prefetching. The addresses are then used to construct a model for which the expected prediction accuracy is calculated. With a high accuracy, the access check enters the *Prefetching* state. If too low, the access check enters the *Waiting* state. In this state, the access check is assigned an age that determines when it will fall back to the *Monitoring* state again.

3.3 Dynamic Adaption of the Prefetching Distance

To reduce the number of messages of an execution, the predictor must emit bulk prefetches that ask for several data items. The number of elements prefetched simultaneously is called the *prefetching distance* N. With an increasing prefetching distance the number of messages may be reduced, but the prediction accuracy generally drops and, thus, DSM protocol activity grows due to unused objects or false sharing. Although $N = 10$ turned out in our measurements to be a reasonable trade-off, a static N is not the best prefetching distance for all applications.

Hence, an automatic adjustment of the prefetching distance N is desired. The local node sends out prefetches to remote nodes and counts how many objects are sent back as an answer (mispredictions are possible, causing the remote node to ignore the request). N is doubled if the number of received objects is higher than 75% of the number of requests. It is decreased by two thirds if less than 25%

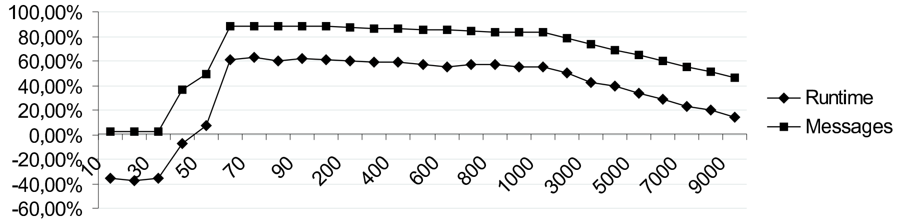

Fig. 3. Effect of the state-transition threshold t on runtime and message count savings.

of the objects arrive. Otherwise, the distance remains unchanged. Over time, N stabilizes for applications with stable memory access behavior.

We tested different techniques that generally converged to the same values but not at the same speed. We also have tested the effect of N if adjusted for each individual access check. The proposed solution with N adjusted globally is the one that gave the best results on average for all benchmarks. For brevity, we only present the best solution, but omit benchmarks.

4 Performance

To evaluate the performance of our automatic optimizer, we measured the performance of four benchmarks. They represent classes of applications with different DSM communication patterns. The selection represents applications that use general-purpose DSMs. We evaluate the benchmarks on a cluster of Xeon 3.2 GHz nodes (2 GB memory, Linux kernel 2.6.20.14, Gigabit Ethernet).

The state-transition threshold t represents the number of executions of an access check required before a decision is made. Fig. 3 shows the effect of t on runtimes and message counts, printing relative savings compared to the uninstrumented 4-node execution. For brevity, we only show results for Blur2D. Small thresholds increase message counts and runtimes, as access checks pollute the prediction model when sent to *Prefetching* too early; many transitions also cause runtime penalties. Increasing t, speed-ups are observed as only beneficial access checks are prefetched. Further increases of the threshold deteriorate performance, as the profiler promotes fewer access checks to *Prefetching*. We use $t = 100$ for the evaluation, since it is the best solution.

Table 1 shows how many access checks are in the codes. Only a few of them actually reach the *Prefetching* state, i. e., the profiling code is replaced by prefetches. In two benchmarks, no access checks were sent to *Waiting*, because they were not executed often enough.

Table 2 lists runtimes and message counts. It shows unmodified benchmarks (*w/o*), with manually added prefetches, with dynamic optimization (*DyCo*), and with automatic distance adjustment (*DynN*). The runtimes of *DynN* are best in most cases. Otherwise they closely match the results of manual prefetching.

SOR iteratively solves discrete Laplace equations on a 2D grid by averaging four neighboring points for a grid point's new value (5,000×5,000 points, 50 iterations). The threads receive contiguous partitions of grid rows and communicate

Table 1. Number of access checks in various states.

	Total number of access checks	No. of access checks that once have reached the *Model Creation* state	No. of access checks that once have reached the *Prefetching* state	No. of access checks that once have reached the *Waiting* state
SOR	10	4	4	0
Water	75	32	27	6
Blur2D	9	3	2	1
Ray	6	2	2	0

Table 2. Runtimes and message counts for the benchmarks (best in bold).

		Runtime (in seconds)				Messages (in thousands)			
	Nodes	w/o	manual	DyCo	Dyn. N	w/o	manual	DyCo	Dyn. N
SOR	2	24.3	24.2	25.5	**23.3**	27.6	6.0	10.4	**5.1**
	4	14.4	**13.5**	13.9	13.8	83.6	17.7	31.6	**14.8**
	6	13.1	12.1	11.6	**11.4**	139.7	30.2	53.3	**26.0**
	8	12.0	11.0	11.1	**10.2**	195.8	42.3	74.2	**34.4**
Water	2	122.6	110.4	99.8	**90.4**	1696.1	1024.7	1079.4	**852.4**
	4	73.0	66.82	63.0	**56.9**	2909.3	1748.7	1774.1	**1354.1**
	6	71.7	56.76	55.8	**48.7**	3639.9	2194.8	2147.9	**1567.8**
	8	66.6	53.47	53.2	**46.0**	4248.0	2543.3	2530.7	**1804.5**
Blur2D	2	10.3	**3.5**	4.6	4.7	224.2	**33.5**	129.2	129.2
	4	6.9	4.1	4.6	**2.5**	386.3	100.3	134.9	**46.2**
	6	8.3	5.3	4.1	**2.4**	484.0	166.1	146.5	**51.5**
	8	10.1	8.3	4.8	**2.7**	583.2	233.2	166.0	**60.6**
Ray	2	44.8	45.3	**44.7**	44.9	9.1	5.9	4.7	**3.1**
	4	22.7	22.6	22.7	**22.4**	27.3	17.1	14.1	**9.3**
	6	15.6	**15.4**	15.6	15.8	45.6	28.6	23.6	**15.6**
	8	13.1	13.0	**12.4**	12.9	64.3	46.3	33.5	**22.1**

reading boundary points of another thread, forming a well-formed, regular data access pattern. Although compilers for array-based languages could statically add prefetches to the compiled code, it is instructive to investigate SOR.

SOR's threads access a small working-set; only four of ten access checks qualify for prefetching. Our automatic system roughly saves 82% of the messages for eight nodes and is slightly better than manual prefetching. Because of the small working-set, a prefetcher cannot improve performance much. Comparing to *DyCo*, the dynamic adaptation of the prefetching distance reduces the message count by 50%. Although all three setups reduce the message count by at least 62% compared to the setup without optimization, runtime is not significantly improved. SOR's computation clearly dominates the average message latency, so that saving messages does not actually pay off in a runtime reduction.

Water is part of the SPLASH benchmark suite [17] and was ported to an object-oriented Java version. Water performs an n-body, n-square simulation of 1,728 water molecules, which are represented by objects that hold the velocity and acceleration vectors. The work is divided by assigning molecules to different threads. Threads repeatedly simulate a single time step of ten (by computing new velocity and acceleration vectors for their molecules) and publish new molecule states by means of a simultaneous update at a collective synchronization point.

Our system selects 32 out of 75 access checks for model creation; only 27 are suited for prefetching. The additional messages (manual vs. DyCo) remain mostly unnoticed in the runtimes. When our optimizer automatically selects the prefetching distance ($N = 70$ instead of $N = 10$), the in-transit messages are reduced by roughly 57% on eight nodes, giving a speed-up of almost 31%.

Blur2D softens a picture of 400×400 points over 20 iterations. The picture is stored as a 2D array of `double`s describing the pixels' gray values. Similar to an SOR stencil, a pixel's value is averaged over the old value and the values of eight neighboring pixels. Prefetching is difficult because the work distribution does not fit the DSM's data distribution scheme. While Jackal favors *row-wise* distribution schemes, Blur2D uses a *column-wise* work distribution. Hence, false sharing and irregular access patterns make Blur2D highly network-bound.

This is directly reflected by Blur2D's poor speed-up behavior. The runtime increases if the node count exceeds four nodes. Blur2D has a very small working-set that makes monitoring access checks difficult. Hence, the manual placement wins for small node counts both in terms of message counts and runtime. For larger node counts the optimizer beats the manual setup. With increasing node counts, Blur2D causes more and more false sharing and the optimizer gathers more data about failing access checks. Thus, it is able to turn off the ones that are too costly. This results in a runtime gain of roughly 73% on eight nodes.

Ray is a simple raytracing application that renders a 3D scene with 2,000 randomly placed spheres. The image is stored as a 2D array of 500×500 RGB values and is divided into distinct areas that are assigned to the threads. As raytracing is embarrassingly parallel, no communication occurs except for the initial distribution of the scenery and the final composition of the finished image. Because of the absence of communication, prefetching should not help much.

Ray's working-set is large enough to allow for message savings. The optimizer identifies two access checks, enabling prefetching for them. This results in a reduction of messages of roughly 65% (48% without adaptation of the prefetching distance). However, this reduction again does not gain any speed-up, as the computational effort completely hides the savings of a few thousands of messages.

5 Related Work

Let us focus dynamic optimizers and prefetching solutions. For brevity, we skip related work in the field of DSM implementation techniques.

Lu et al. [12, 13] implemented a dynamic optimizer that inserts prefetching instructions into the instruction stream. Using Itanium performance counters, the system works on available hot trace information and detects delinquent loads in loop nests. Our implementation detects hot traces automatically as it starts from monitoring access checks and only replaces access checks that are likely to benefit from prefetching; explicit hot trace information is not needed.

Dynamic frameworks such as DynamoRIO [4] interpret the application code at runtime, search for hot traces, and optimize the application. UMI [18] uses DynamoRIO to implement a lightweight system that profiles and optimizes programs. DynamoRIO selects program traces and UMI gathers memory reference profiles. The profiles are used to simulate the cache behavior and to selects delinquent loads. UMI also integrates a simple stride prefetching solution into the framework. Since these optimization systems are trace-based systems, an optimization leads to modifications of a set of basic blocks. In contrast, our system handles each access check independently. This enables our system to de-instrument individual access checks that are not profitable and to avoid performance losses due to monitoring single, unprofitable access checks.

Chilimbi/Hirzel's framework [5] samples the program's execution to decide what portions of the code should be optimized. Using the Vulcan [15] editor for IA-32 binaries, it creates two versions of each function. While both contain the original code, one also contains instrumented code and the other is augmented with instrumentation checks. A state machine decides the state of functions at runtime. To lower the overhead, states are globally frozen. We also use state freezing to avoid thrashing of access checks by keeping states fixed over time. We employ state machines to switch between different types of access checks, but we consider each access check independently. This gives finer control over which access checks undergo state transitions and which states are kept fixed.

Finally, being a well-known technique, we only cover a short selection of prefetching techniques for S-DSMs. Adaptive++ [3] and JIAJIA [11] use lists of past memory accesses to predict, while Delphi [14] uses as a prediction table a hash over the last three accesses. An inspector/executor pattern determines future accesses in an OpenMP DSM [8]. These predictors prefetch asynchronously, which gives not enough overlap to hide the high network latencies in object-based DSMs. Stride predictors [7] do not fit either, as they cannot handle the complex memory access patterns of object-based DSMs (which also is a problem for page-based predictors). In contrast to our system, the predictors cannot temporarily be turned off if the prediction accuracy drops.

6 Conclusions

We have shown that it is worthwhile to integrate an automatic dynamic optimizer into object-based S-DSMs. With binary rewriting techniques, superfluous or unprofitable monitoring or prefetching calls can be removed. Measurements show that, on average, performance improves by 18% when binary rewriting based on a state machine is used to automatically place the prefetching access checks; the message count drops by 52%. The dynamic adjustment of the prefetching distance saves 26% of the runtime and decreases the number of messages by 70%. In total, we have achieved runtime improvements of up to 73% on the benchmarks.

We are currently working on an automatic system that automates every prefetching aspect, from the choice of the predictor to the values of the prefetch distance and the threshold value, thus becoming totally transparent to the user.

References

1. TOP500 List. http://www.top500.org/, November 2007.
2. R. Begleiter, R. El-Yaniv, and G. Yona. On Prediction Using Variable Order Markov Models. *Journal of Artificial Intelligence Research*, 22:385–421, 2004.
3. R. Bianchini, R. Pinto, and C.L. Amorim. Data Prefetching for Software DSMs. In *Intl. Conf. on Supercomputing*, pages 385–392, Melbourne, Australia, July 1998.
4. D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Intl. Symp. on Code Generation and Optimization*, pages 265–275, San Francisco, CA, March 2003.
5. T.M. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation*, pages 199–209, Berlin, Germany, June 2002.
6. MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, MPI Forum, July 1997.
7. 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.
8. W.-C. Jeun, Y.-S. Kee, and S. Ha. Improving Performance of OpenMP for SMP Clusters through Overlapping Page Migrations. In *Intl. Workshop on OpenMP*, Reims, France, June 2006. CD-ROM.
9. D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.
10. M. Klemm, J.C. Beyler, R.T. Lampert, M. Philippsen, and P. Clauss. Esodyp+: Prefetching in the Jackal Software DSM. In *Proc. of Euro-Par 2007*, pages 563–573, Rennes, France, August 2007.
11. H. Liu and W. Hu. A Comparison of Two Strategies of Dynamic Data Prefetching in Software DSM. In *15th Intl. Parallel & Distributed Processing Symp.*, pages 62–67, San Francisco, CA, April 2001.
12. 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 *36th Ann. IEEE/ACM Intl. Symp. on Microarchitecture*, pages 180–190, San Diego, CA, December 2003.
13. J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. *Journal of Instruction-Level Parallelism*, 6, April 2004. Online.
14. E. Speight and M. Burtscher. Delphi: Prediction-Based Page Prefetching to Improve the Performance of Shared Virtual Memory Systems. In *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 49–55, Las Vegas, NV, June 2002.
15. A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.
16. R. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang, and H.E. Bal. Runtime Optimizations for a Java DSM Implementation. In *ACM-ISCOPE Conf. on Java Grande*, pages 153–162, Palo Alto, CA, June 2001.
17. S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd Intl. Symp. on Computer Architecture*, pages 24–36, St. Margherita Ligure, Italy, June 1995.
18. Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous Memory Introspection. In *Intl. Symp. on Code Generation and Optimization*, pages 299–311, San Jose, CA, March 2007.