A Meta-predictor Framework for Prefetching in Object-based DSMs

Jean Christophe Beyler\textsuperscript{1}, Michael Klemm\textsuperscript{2}, Philippe Clauss\textsuperscript{3}, and Michael Philippsen\textsuperscript{2}

\textsuperscript{1}Department of ECE ● University of Delaware
Newark, DE 19711 ● USA
beyler@capsl.udel.edu,
\textsuperscript{2}Computer Science Department 2 ● University of Erlangen-Nuremberg
Martensstr. 3 ● 91058 Erlangen ● Germany
\{klemm, philippsen\}@cs.fau.de,
\textsuperscript{3}Université Louis-Pasteur de Strasbourg ● ICPS-LSIIT
Pôle API, Bd Sébastien Brant ● 67400 Illkirch ● France
clauss@icps.u-strasbg.fr

SUMMARY
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 on clusters. In DSM systems, reducing the number of messages between cluster 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 monitor the access behavior of the application and place prefetches where they are beneficial to speed up the application. In addition, we use two distinct predictors to handle different types of access patterns. A meta-predictor analyzes the memory access behavior and dynamically enables one of the predictors. Our system also adapts the number of prefetches per request to best fit the application’s behavior. The evaluation shows that the performance of our system is better than manual prefetching. The number of messages sent decreases by up to 90%. Performance gains of up to 80% can be observed on benchmarks.

1. Introduction

The high-performance computing landscape is mainly shaped by clusters, which make up 82% 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

Received December, 06th 2006

Copyright © 2000 John Wiley & Sons, Ltd.
a globally shared address space, alleviating the need to place explicit calls to a message passing library that handles communication (e.g. MPI [7]). Instead, a middleware layer accesses remote memory and ensures memory coherence. Examples of such S-DSM systems are JIAJIA [12], Delphi [15], or Jackal [17].

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 tries to solve this performance issue 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. If the fetch is executed at the right time, data arrives just-in-time when needed by the program. Based on heuristics, compilers statically analyze the code and add prefetches to it 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 [17] adds monitoring code to the executable that profiles memory accesses. A profiler classifies the memory accesses and enables prefetches if beneficial. Dynamic code rewriting keeps 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. We have implemented two prefetchers in this system. First, a stride predictor [8] is enabled if it achieves a sufficient accuracy. For more complex data access patterns, the Esodyp+ predictor [11] is used. It tracks memory access patterns and prefetches future memory accesses. While the original Esodyp+ framework 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.

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 memory consistency model of the programming language has to be respected, which involves invalidating or updating replicas of objects on other nodes. Testing for object availability can only be implemented in software.

In general, an S-DSM cannot use specialized hardware for object access checks, as it should
int foo(SomeObject o) { return o.field; }

int foo(SomeObject o) {
  if (!readable(o))
  fetch(o, readable);
  return o.field;
}

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

leaq -1308622848(%r8), %rdi
shrq $5, %rdi
movq %rdi,%rcx
movq %fs:(0), %rdx
movq thread_local_dsm_read_bitmap@TPOFF(%rdx), %rdx
bt %rcx, (%rdx)
jc .L28961
movq %rax, %rdi
movq %rdi,%rax
.L28961:

Figure 2. Example of an access check in assembly code, without prefetching.

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 restricted pages are accessed, the OS triggers the S-DSM, notifying it about the faulty access. After the S-DSM has loaded the page, the access is re-issued and the application continues. Delphi [15] and others [4, 9, 12] use this approach to implement a page-based DSM. However, memory protection can only be applied at the page level and renders this option unusable for object-based S-DSMs, as it causes false faults on objects that reside on the same page. Hence, for such S-DSMs, the access checks has to be implemented in software, which is easy to do, because it often requires only a single bit test.

We use the Jackal object-based DSM [17] for our prototype implementation. Jackal compiles a Java program to a native executable, e.g. for IA-32 or Itanium, and inserts access checks into the code to prepare it for the S-DSM environment. 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 message is sent to the home node that stores the object’s master copy. In turn, this request is answered with the requested data. This protocol causes a delay of roughly two times the network latency plus the cost of serializing and de-serializing the object’s data.

Fig. 2 shows the assembler fragment of the read test of Fig. 1 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. The runtime requests the object data from the object’s home node and waits until data has arrived.

Jackal uniquely identifies each object in the DSM space by a so-called Global Object Reference (GOR). A GOR is a tuple consisting of the logical rank of the object’s home node
and the object’s address on that node. The GOR does not change over the lifetime of the object; only if the object is reclaimed by the garbage collector, the GOR is released and may be recycled. When non-local objects are received, the DSM runtime allocates space in a local caching heap and assigns a local address to the received objects for efficient local access. The DSM runtime maintains a mapping of local addresses to their corresponding GORs and hence to the home nodes (for status and data updates).

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 that are rarely executed or that exhibit random behavior, a classification by the profiler is crucial for the efficiency of the optimizer. There are several ways how 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 \texttt{switch} statement that decides between monitoring, prefetching, and no prefetching, the guards cause performance losses that result from additional instructions, increase load on the memory bus, and 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. This code is then either replaced with prefetching code or it is removed if prefetching is not beneficial. To remove code, the rewriter has to replace the code with \texttt{nop}s, as the subsequent code cannot replace the to-be-removed code fragment (moving code implies checking and modifying most branch instructions). However, \texttt{nop}s pollute the instruction cache and pipelines and 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 from 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, 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 \texttt{switch} statements or \texttt{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 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 or not. Fig. 3 shows the state machine.

Monitoring is the initial state for access checks. If an access check is often executed, it is sent to the Model Creation state (reaching a threshold \( t \) of invocations). 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. Waiting represents the state in which the original access check code without instrumentation is executed (the other states use instrumented versions). To detect changes in the application’s behavior that might render prefetching beneficial, access checks are periodically re-instrumented. To reduce the overhead and to avoid state thrashing, the time between re-instrumentations increases at each cycle.

Prefetching is 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 two predictors. First, we use a Stride predictor \([8]\) that calculates the stride, the difference between the two accessed addresses, and uses it to predict future addresses. If the accuracy of the Stride predictor is below a certain threshold, the second predictor is used. This second predictor is the Markov model predictor Esodyp+ \([11]\), 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 \([2, 10]\).
The profiler needs to efficiently identify access checks for profiling. Since Jackal uses function calls for access checks (see Fig. 2), state transitions rewrite these function calls. We use the call’s return address as a key into a hash table that stores the profiling data. In other DSM systems, the compiler could tag 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 linear 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.

First, these addresses are considered for stride prefetching. If the access pattern is not stride-based, the addresses are then used to construct an Esodyp+ model for which the expected prediction accuracy is calculated. With a high accuracy with either predictor, the access check enters the Prefetching state and will use that predictor for prefetching. If neither predictor has a high accuracy, the access check enters the Waiting state. In this state, the access check is assigned an age that determines when it will return to the Monitoring state.

3.3. Dynamic Adaption of the Prefetching Distance

To reduce the number of transmitted DSM messages, 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 in our measurements $N = 10$ turned out 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% of the objects arrive. Otherwise, the distance remains unchanged. Over time, $N$ quickly 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 over all benchmarks. For brevity, we only present the best solution in the performance section.

4. Stride Predictor and Esodyp+

In this section, we present two predictors (a stride predictor and Esodyp+). We then show how a meta-predictor determines which of the two predictors is enabled.

4.1. Stride Predictor

Our multi-stream stride predictor implementation [8] calculates a new stride as the difference between the current address and the last address. 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 quickly enough in the sequence of strides. To further increase the chances of predictions, if the predictor notices too many miss-predictions, it first tries to see if, using a newly calculated stride, the predictor would no longer mispredicts. If this is not the case, the predictor sends the access check back to the Monitoring state. 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 predictor often mispredicts in non-array programs or for complex data distributions.

4.2. Esodyp+

The Esodyp+ predictor [11] extends Esodyp (Entirely Software DYnamic data Prefetcher), a predictor that modelizes address sequences with a variation of a classic Markov model [3]. This section briefly presents the model, how it reacts to the addresses passed by the DSM framework, and how it helps to prefetch objects. A complete discussion can be found in [3].

Classical Markov predictors [10] 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 how much items will be predicted; a value of 1 means that one next element is predicted. 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 [10].

To illustrate graph creation, let us assume Esodyp+ tracks the following sequence of addresses: A, B, C, B, A, B, C. Fig. 4(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. 4(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. 4(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, Esodyp+ predicts in constant time. This is a major benefit compared to other predictors that need to perform calculations in order to predict.

However, for each element of the considered sequence, at least one node in the graph is created. Since this makes 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 addresses and store only the difference in the model.

Originally, Esodyp was implemented using a construction phase to create the model that is later used in the prediction phase. Esodyp+ merges both phases and emits predictions even while it is constructing. This helps 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 [3].

4.3. The Meta-predictor

There are various reasons to have multiple predictors in a dynamic prefetching system. It has been shown in [11] that different predictors perform differently depending on the access pattern. Therefore, it is beneficial to choose between predictors at runtime. It is important to notice that each access is handled independently in this predictor choice.

Our meta-predictor first enables the stride predictor for prefetching. For each access check that enters the Model Creation state, the access pattern is first checked for a stride pattern. If the stride predictor cannot accurately predict future accesses, the Esodyp+ predictor is enabled. If Esodyp+ does not consider the pattern predictable, the access check is sent to the Waiting state.

5. 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 use a cluster of Xeon 3.2 GHz nodes (2 GB memory, Linux kernel 2.6.20.14, Gigabit Ethernet) for benchmarking.

5.1. Esodyp+

Table I 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 I. Number of access checks in various states (Esodyp+ only).

<table>
<thead>
<tr>
<th>Model Creation</th>
<th>Prefetching</th>
<th>Waiting</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Water</td>
<td>75</td>
<td>32</td>
</tr>
<tr>
<td>Blur2D</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Ray</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Table II. Runtimes and message counts for the benchmarks (Esodyp+ only, best in bold).

<table>
<thead>
<tr>
<th>Nodes</th>
<th>w/o manual</th>
<th>DyCo</th>
<th>Dyn. N</th>
<th>w/o manual</th>
<th>DyCo</th>
<th>Dyn. N</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>2</td>
<td>24.3</td>
<td>25.5</td>
<td>23.3</td>
<td>27.6</td>
<td>10.4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>14.4</td>
<td>13.5</td>
<td>13.9</td>
<td>13.8</td>
<td>17.7</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>13.1</td>
<td>12.1</td>
<td>11.6</td>
<td>11.4</td>
<td>30.2</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12.0</td>
<td>11.0</td>
<td>11.1</td>
<td>10.2</td>
<td>42.3</td>
</tr>
<tr>
<td>Water</td>
<td>2</td>
<td>122.6</td>
<td>99.8</td>
<td>90.3</td>
<td>1096.1</td>
<td>1024.7</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>73.0</td>
<td>66.82</td>
<td>56.9</td>
<td>2909.3</td>
<td>1748.7</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>71.7</td>
<td>56.76</td>
<td>55.8</td>
<td>3639.9</td>
<td>2194.8</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>66.6</td>
<td>53.47</td>
<td>53.2</td>
<td>4484.0</td>
<td>2543.3</td>
</tr>
<tr>
<td>Blur2D</td>
<td>2</td>
<td>10.3</td>
<td>3.5</td>
<td>4.6</td>
<td>4.7</td>
<td>224.2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6.9</td>
<td>4.1</td>
<td>4.6</td>
<td>2.5</td>
<td>386.3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>8.3</td>
<td>5.3</td>
<td>4.1</td>
<td>2.4</td>
<td>484.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10.1</td>
<td>8.3</td>
<td>4.8</td>
<td>2.7</td>
<td>583.2</td>
</tr>
<tr>
<td>Ray</td>
<td>2</td>
<td>14.8</td>
<td>45.3</td>
<td>44.7</td>
<td>44.9</td>
<td>9.1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>22.7</td>
<td>22.6</td>
<td>22.7</td>
<td>22.4</td>
<td>27.3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>15.6</td>
<td>15.4</td>
<td>15.6</td>
<td>15.8</td>
<td>45.6</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>13.1</td>
<td>13.0</td>
<td>12.4</td>
<td>12.9</td>
<td>64.3</td>
</tr>
</tbody>
</table>

Table II 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 meta-predictor is turned off and the system only relies on Esodyp+ as the predictor. The transition threshold was set to 100 for the experiments (see Section 5.3). 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). A thread receives a contiguous partition of the grid rows and communicates reading boundary points of two neighboring threads, forming a well-formed, very 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 out 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. Compared 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 [18] 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 $400 \times 400$ points over 20 iterations. The picture is stored as a 2D array of `doubles` 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 $500 \times 500$ 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.

### 5.2. Meta-predictor

Table III lists the runtime and message counts with the meta-predictor enabled for the benchmarks. For reference, the table also contains the results of the unmodified benchmarks.
Table III. Runtimes and message counts for the benchmarks with meta-predictor (best in bold).

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ACs using Stride</th>
<th>Runtime (in seconds)</th>
<th>Messages (in thousands)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>w/o</td>
<td>Dyn. N</td>
</tr>
<tr>
<td>SOR</td>
<td></td>
<td>2</td>
<td>8/8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>15/15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>25/25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>30/30</td>
</tr>
<tr>
<td>Water</td>
<td></td>
<td>2</td>
<td>17/71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>32/168</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>49/212</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>51/289</td>
</tr>
<tr>
<td>Blur2D</td>
<td></td>
<td>2</td>
<td>6/6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1/12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>1/18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1/24</td>
</tr>
<tr>
<td>Ray</td>
<td></td>
<td>2</td>
<td>4/4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>8/8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>12/12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>16/16</td>
</tr>
</tbody>
</table>

and the Esodyp+ predictor with automatic distance adjustment (DynN). The meta-predictor uses the same automatic distance adjustment as the Esodyp+ version.

The third column of Table III shows the number of access checks (ACs) that are sent to the stride predictor. It is the sum of the number of access checks for every node. For example, if we consider Water, 32/168 denotes that out of 168 access checks for all the nodes, only 32 were sent to the stride predictor. Instead of showing averages like in Table I, we show total sums since, depending on the application and the node count, each node can have a different behavior, as it is shown for SOR, Water, or Blur2D. Averages would hide these variations. For both Water and Blur2D, these numbers vary significantly depending of the number of nodes.

For SOR and Ray, the accesses are always stride-based for any node configuration. SOR only accesses arrays in a regular fashion and thus gives the stride predictor the chance to predict accurately. During allocation of Ray’s input data, the DSM runtime creates the objects in consecutive memory location, effectively causing stride-based patterns when accessing these objects. The number of messages are reduced by 25% and 8%, respectively, compared to the DynN version. However, for Blur2D and the 2-node execution, every access check is considered stride based. Hence, the meta-predictor significantly reduces the number of messages and achieves a considerable reduction of the execution time.

The key difference between the results for DynN and Meta in a 2-node scenario for Blur2D is how each predictor handles miss-predictions. In the case of the stride predictor, a miss-prediction does not modify the internal state of the predictor and it continues to prefetch using the last calculated stride. Although it sends the access check back to the Monitoring state if too many miss-predictions occur, in this case, it simply ignores the rare error cases and continues to prefetch correctly.

On the other hand, when considering the Esodyp+ predictor, the state of the predictor is important. When a miss-prediction occurs, the whole state of the predictor must be re-initialized. Although this only takes a limited number of accesses, the predictor does not automatically re-issue prefetches, even in the case of a strongly strided access pattern. It is
for these reasons that the stride predictor obtains such a good performance on the 2-node configuration for Blur2D. For the executions with other node counts, the stride predictor fails to get a good accuracy, and therefore Esodyp+ is used. Finally, this explains why for more than 2 nodes the results are similar between both versions.

Finally, for every execution, the meta-predictor outperforms the DynN version in terms of execution time and number of messages except for Blur2D where the stride predictor is not used for higher than two nodes. Although a Markovian predictor can also predict stride-based patterns, it needs a warm-up period and if there is noise in the sequence, the predictor will need to re-adjust itself accordingly. Hence, for stride-based patterns a stride predictor is less fragile and achieves a better performance. Compared to the DynN version, there is an average 13% improvement with respect to the number of messages of the system with an 18% and 20% improvement for Blur2D and SOR, respectively. Runtime is only significantly improved for Blur2D with a 55% reduction (compared to DynN). Compared to the execution time without optimization, there is a gain of 80%.

5.3. Transition Threshold

The state-transition threshold \( t \) represents the number of executions of access checks that is required before a decision is made. Fig. 5 shows the effect of \( t \) on runtimes and message counts. It depicts Esodyp+’s relative savings compared to the uninstrumented 4-node execution of each benchmark (see Section 5.1). On average, we notice that too small thresholds deteriorate the performance and increase message counts, as access checks pollute the prediction model when sent to Prefetching too early; many transitions cause additional runtime penalties. When it is increased, the speed-ups improve, as only beneficial access checks are prefetched. Further increases of the threshold again deteriorate performance, as the profiler promotes fewer access checks to the Prefetching state. This can be seen for SOR or Raytracer where there is no longer any performance gain. Also, the runtime is not significantly reduced for these two programs since there is a small number of messages. Thus, the reduction of the number of messages, although it reaches 80% and 68% respectively, does not have a significant impact on the runtime. On the other hand, for programs such as Water and Blur, it is important to choose the correct threshold value to obtain the best performance gains. We used \( t = 100 \) for the evaluation, since on average it is the best solution as a unique value for each benchmark program.

6. Related Work

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

Lu et al. [13, 14] 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.
Figure 5. Effect of the state-transition threshold $t$ on runtime and message count savings.
Frameworks such as DynamoRIO [5] interpret the application code at runtime, search for hot traces, and optimize the application. UMI [19] 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 select 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 both 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 [6] samples the program’s execution to decide what portions of the code should be optimized. Using the Vulcan [16] 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 states are kept fixed.

As prefetching techniques for S-DSMs are well-known, we only cover a selection of prefetching techniques for S-DSMs. Adaptive++ [4] and JIAJIA [12] use lists of past memory accesses to predict, while Delphi [15] uses as a prediction table a hash over the last three accesses. An inspector/executor pattern determines future accesses in an OpenMP DSM [9]. These predictors prefetch asynchronously, which gives not enough overlap to hide the high network latencies in object-based DSMs. Stride-only predictors [8] 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.

7. Conclusions

We have shown an automatic dynamic optimizer for object-based S-DSMs. With binary rewriting techniques, superfluous or unprofitable monitoring or prefetching calls can be removed. Using a meta-predictor to enable either a stride predictor or the Esodyp+ predictor, we are able to reduce the number of messages and speed-up applications that represent different classes of applications with different DSM communication patterns. Measurements show performance improvements by 18% on average when binary rewriting with our state machine automatically places the prefetching access checks; the meta-predictor reduces the message count by 59%. 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 80% on some benchmark programs. These results were obtained by creating a meta-predictor that uses a stride predictor to handle stride-based access checks and the Esodyp+ predictor for more complex access patterns. Both types of patterns can be handled at a low overhead, making the system efficient in a dynamic framework.
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