ESODYP: An Entirely Software and Dynamic Data Prefetcher based on a Memory Strides Markov Model

Abstract

Many works have shown that data prefetching can be an efficient answer to the well-known memory bottleneck. Although several approaches ranging from dynamic hardware to static software mechanisms have been proposed, no pure and stand-alone dynamic software data prefetching solution has yet been proposed. However such a portable approach can be quite worthy regarding the number of data-intensive applications having a dynamically changing memory behavior and regarding the ever growing variety of processor and memory architectures.

In this paper, we propose an Entirely SOftware and DYnamic data Prefetcher (ESODYP) based on a memory strides Markov model. It runs in two main phases: a short training phase where a graph coding sequences of occurring memory strides is constructed, and an optimizing phase where predicted addresses are prefetched while some informations in the graph are updated by continuously monitoring the program.

Significant speed-ups are obtained on an Itanium-2 processor using ESODYP for several benchmark programs that could not have been optimized statically. It is particularly shown that the

induced software overhead can represent a minor execution time regarding performance improvements, due to a careful lowering of the optimizer computations and memory accesses.

This work also opens some interesting perspectives in the development of pure software dynamic optimizers.

1 Introduction

It is quite well-known that memory causes a serious performance bottleneck in spite of the use of caches, since the implemented loading and replacement strategies are unable to suit all possible program memory behavior. Many works have shown that software controlled policy of hardware mechanisms can significantly improve their efficiency. A compiler can be able from a static analysis of the source code to generate some instruction hints [4]. However, such an approach is only exploitable for static control and data structures as for-loops accessing multi-dimensional arrays through affine reference functions. When considering more general control structures accessing data through pointers, static optimization can generally not apply since the essential information are not known at compile-time and can only be observed during execution. Hence dynamic analysis and optimization have become an important area of research.

Works focusing on a dynamic approach can be distinguished by the presence of an off-line phase, and by the hardware or software portion of the proposed system. An off-line phase allows to avoid the overhead induced by a necessary training step of the implemented optimization strategy. However, such a phase can be demanding from multiple execution profilings and program-specific analysis. For example in [16, 13], cache-miss detection is done off-line in order to create threads for data prefetching on an hyperthreaded processor [10].

A fully on-line system is more challenging, since the whole overhead has to be more severely reduced. But no initial profilings are necessary and the system is transparent to the user.

Also pure hardware solutions allow to avoid the overhead of a software system [12, 7, 14, 18, 17, 2, 1]. However, those cannot have the flexibility of software to implement sophisticated strategies and are obviously not portable. Hybrid systems can provide nice solutions but are also not portable to any processor architectures [16, 15, 13]. The usage of operating system specific tools also reduces the range of possible target platforms. For example in [6], the presented software optimizer uses the software tool *VULCAN* only available on *Windows* platforms.

Most hardware models [12, 14] rely uniquely on addresses yielding cache misses. To achieve this, they rely on an on-chip buffer added to the processor used to calculate what is prefetched. A pure software solution does not use an on-chip buffer. Most pure software approaches [3, 6, 9, 5, 15] consider a large memory block shared by the original process and the optimizer. Moreover in a pure software system, it is not known whether a data access induces a cache miss or a cache hit. For example, the system *ADORE* [15] detects cache misses using the *Itanium* hardware counters, thus it can be considered as hybrid.

In this paper, we propose a pure software and fully on-line system based on a memory strides Markov model. Classically, a Markov predictor uses a history of accessed data to predict the next data that is going to be accessed by a program [12, 14]. Using statistical information on past memory behavior, it tries to load the data before the program effectively needs it. If done correctly, this can result in a significant speedup.

In our model, we do not consider the accessed data memory addresses but the strides occurring between successive accesses. Unlike other approaches, a large amount of accessed memory can therefore be considered while modeling a relatively constant memory behavior of the program. Hence in many cases a moderatly small amount of memory is needed to contain all the information used to achieve prefetching. Moreover, our model considers prediction from any number of past occurring values unlike other Markov models only considering a unique past value [12, 14, 18]. Our experiments show significant speedups for several benchmark programs using our dynamic optimizer.

In the next section, we present our Markovian model and show how it fits in a dynamic approach. Section 3 details ESODYPS's two main phases: the training phase consisting in the construction of a graph and the prediction phase consisting in prefetching the predicted memory accesses. ESODYP's usage and several experiments are presented in section 4. Finally, conclusions are given in section 5.

2 The Markov predictor

Our Markovian predictor remembers sequences of strides between successive data accesses. When the predictor is given a new address, it tries to map the new stride computed from the previous access and the previously occurring strides to a particular sequence. If successful, it is able to predict the next access. Accuracy of the prediction depends on the amount of information stored by the predictor and on the relative redundant behavior of the program memory strides.

The maximum number of past values stored by the predictor is called the *depth*. Most models [12, 18] use only a one-depth predictor. Our model considers any depth and hence can give better predictions as shown in our experiments.

Considering memory strides instead of actual addresses can give the predictor a better chance to keep all the information retrievable by the program with the least memory necessary. Of course, this strategy is not universally the best. If the number of different strides is too high, our model uses a lot of memory. But it is able to capture many regular memory behaviors often occurring in programs, unlike methods based on actual adresses whose applicability is limited to behaviors characterized by a lot of temporal data reuses.

We call *prefetching distance* the number of strides we attempt to predict in advance. Suppose we know that after a stride sequence S the predicted strides are A, B and C. Then it is possible to prefetch addresses $b+\sum S+A$, $b+\sum S+A+B$ or $b+\sum S+A+B+C$, where b denotes the base address of the accessed data and $\sum S$ denotes the sum of all strides in the sequence S. Since prefetching takes a certain amount of time, if address $b+\sum S+A$ is prefetched, it is possible that the program arrives at the point where the concerned data is needed before the prefetch has been completed. The program would then stall. In such a case, it would be wrong to prefetch address $b+\sum S+A$ and it would be preferable instead to prefetch addresses $b+\sum S+A+B$ or $b+\sum S+A+B+C$ as it is done in [15].

When the predictor gets the current accessed address, it must decide what to predict. In the most favourable case, there is only one possible choice. In the case of several choices, the most naive answer is to do the prefetch from each possibility. But since we are going to predict two or three strides in advance, we obviously cannot predict every possible address. Moreover most processors support a limited amount of simutaneous prefetches (16 on the Itanium-2 [11]). Because of both factors, we must restrain the number of prefetches at each step by prioritizing all the possibilities as it is explained while presenting the model creation process in the next section.

The predictor depth influences considerably the prediction accuracy. Each program can be characterized by the necessary depth yielding a significant speedup. Obviously, a higher depth gives a better chance for accurate predictions but needs more memory to store the information. Therefore we need to determine what is more essential in each situation between depth and memory in order to get the best speedup. As an illustration of this fact, our experiments in subsection 4.2 point out the `ks` program from the *Pointer Intensive Benchmarks* which requires a depth of 2 for an actual speedup to be observed.

Like other dynamic optimizers [3, 9, 15], our model does not try to compete with static optimizations of compilers. Its purpose is to reduce memory contention for data accesses that could not have been optimized at compile-time. For example, we do not consider array accesses through affine reference functions in for-loops, since such memory references can be efficiently considered statically giving evidently better results than any dynamic process could do.

One main challenge of a dynamic optimizer is to keep its overhead low enough so that the gain makes the overall program run faster.

There are different ways for implementing a dynamic optimizer:

- If available, a second processor can calculate all the information needed for the predictions.

- The optimizer can be implemented as a second thread giving the chance to mask the processing time as in the previous case. Such an approach can be efficient using a hyperthreaded processor [13, 10].

- All the code can be left in one unique thread executing either our optimizer or the original program.

Notice that a second thread would require synchronizing mechanisms likely raising the overhead. Our implementation uses the third approach not relying on a particular processor architecture.

Our model is not yet totally transparent at this time since it still needs some help from the programmer. A function links the original program to the optimizer. This function feeds our model with accessed addresses and automatically prefetches the predicted data. A more detailed description of this API is given in the next section.

Since the actual accessed addresses are known, it is possible to monitor the number of correct predictions by comparing them with the prefetched addresses. The optimizer could be stopped whether this number exceeds a certain threshold. By this way, the speed-down that would most likely occur would be reduced.

3 Implementation

In models that only need the previous address for prediction, the use of a table seems natural. Some models [12, 14] use such tables storing more than one prediction giving the possibility to prefetch more than one address.

Since we use sequences of strides with different depths, our model does not use a table. However our data structure has to allow the prediction process to be as fast as possible. We use a graph to represent the different sequences that have already been monitored. Each time a new stride is caught, the next ones can be predicted with a complexity not related to the depth used.

To illustrate the graph creation phase, let us consider the following example of a sequence of strides that a running program could have given our model: 1, 2, 16, 2, 32, 2, 16, 2, 32.

Previous strides	Next stride
2	32, 16
1 2	16
2 16	2
16 2	32
2 32	2
32 2	16

Table 1: A prediction table

Table 1 shows what would be stored if the sequence was coded in a table with a maximum depth of 2. The left column represents the last strides made by the program memory accesses and the right column the next stride that is probably going to occur, with the assumption that what occurred previously is likely to be repeated.

We notice that after a stride of 2, the next stride is uncertain as it is not known what stride occurred before. In our case, the probabilities that a stride of 16 or 32 occurs are the same. However it is likely that sometimes a certain stride follows more often a certain sequence. It is this most probable stride which is selected for prediction.

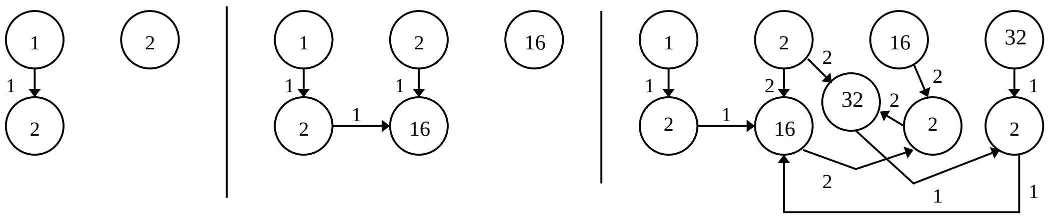

Figure 1: The Markovian graph (with a depth of 2) after two strides (a), three strides (b) and the full sequence (c) has been passed to the model

To explain how the graph works, let us see what happens as the two first strides are caught (Figure 1 (a)). Similarly to the table, we restrict ourselves to a depth of 2. The meaning of the single edge is that after a stride of 1, to our knowledge, a stride of 2 will occur. In the case of nodes without successor, it is not known what could happen next. For example, the connex group (1,2) is not attached to anything since we do not have any information on the next stride. The node 2 alone symbolizes that, if all we know is that there was a stride of 2, nothing can be predicted. The edge label 1 means that this edge has been followed once. Such labels are used to select the most followed edges.

We can see how the graph construction evolves on figure 1 (b) while another stride is added to our model. Two more nodes have been added to the graph. The first node 16 is attached to both nodes 2. This symbolizes that after a 2, and after the sequence (1,2) as well, a 16 occurs. The second node 16 tells that, knowing only that the last stride was a 16, what happens next is not known.

Eight nodes are used in the graph when the whole sequence has been processed on figure 1 (c). Now some edge labels have value 2 meaning that those edges were followed twice.

Our graph has many similarities with suffix trees or suffix automata constructed for pattern recognition algorithms in strings [8]. But since we have quite different usage requirements, our graph can be seen as a mix between a suffix tree and a suffix automaton.

This graph construction process has to stop after a while as the graph starts to be used in the prediction phase. As the construction is stopped, the optimizer points to the last created node and starts prediction. When receiving a new stride, it checks whether there is an edge from the current node leading to that stride. If so, the pointer to the current node is updated. Otherwise two strategies are considered:

- If a root node for the received stride exists, that node becomes the current node as no information about the past occurring strides is known and a prediction can be made.
- If no root node for the received stride exists, nothing is done until receiving the next stride that hopefully will allow prediction.

Searching for a root node associated to a given stride is a frequent operation that therefore has to be fast. That is why a binary search tree is added to our graph.

Notice that like other optimizers using the concept that programs act in different phases [3, 9, 15], it is possible to construct an associated graph at each phase beginning. This approach reduces the memory consumption of the optimizer and therefore also reduces its overhead. In our optimizer, phase detection is done by monitoring the number of consecutive miss predictions. When a given threshold is reached, the graph is flushed and the construction is restarted.

Another solution would be to simply restart the construction without flushing the graph. Eventhough in some cases it might seem opportune to do so, we think that in general a phase change yields a quite different memory behavior. Some edge labels would include the previous phase and so would have higher counts than the recently added edges. It would take a certain amount of time before these latter edges would be followed, yielding a lot of misspredictions. If the construction is started over, the counters are reset and the number of nodes does not grow unnecessarily.

As said in section 2, a choice between different possible predictions has to be made and prediction of a few strides ahead of time can be necessary. But since we cannot afford to figure out what stride is to be made at each step, this stride is precalculated. Hence the sum of all strides to be made is directly known when arriving at each node.

The full stride has also to be updated as the prediction changes. Suppose we are on the root node 2 in Figure 1 (c), 32 is the best prediction and only two strides ahead are prefetched. Consequently the node 2 has a variable saying "prefetch 32+2". If 16 becomes the best prediction then this variable must be changed to "prefetch 16+2". Considering that the number of changes is not too important, the precalculation overhead can be quite small. In the worst case scenario, the optimizer would have to calculate the stride at every access, which would have been the case without any precalculation.

Each node owns a list of its successors, that is to say the strides that have followed the current sequence. To reduce the overhead, a pointer to the most probable node is used. To reduce it even more, a LRU management strategy of the last two nodes accessed after the current node has been implemented. The objective is to minimize the number of times the list of successors of a node has to be scanned.

A global data structure holds all the information needed by the model: the predictor depth, the prefetching distance, the graph, the threshold of consecutive misspredictions, the function pointer *fct*,...

Our dynamic optimizer is composed of a few functions. It will automatically prefetch the data once it has a graph completely constructed during the training phase. The different functions are shown in Table 2. Function `fct` is not really a function but a pointer accessed through the Markov structure. If different sequences in the same program have to be monitored, then it is possible for each sequence to have different parameters.

Function name	Description
`initialize`	Initializes the Markov structure
`set_param`	Lets us parameter the depth of the construction, distance of prefetching...
`set_address`	Lets the programmer update the base address, since the model uses strides
`fct`	The main function that is the link between the main program and our model
`clear`	Clears the structure

Table 2: The functions used by the programmer

It is through *fct* that three functions are called depending in which of the phases the model is in. At the beginning, `fct` points to a small function that is only there to set the base address. Since strides are considered, the offset has to be known before creating the graph. After the first call, `fct` points to the construction procedure. Once the construction is finished, it points to the prediction procedure: prediction, followed edge labels incrementation and full stride variable update if necessary.

The construction procedure has a life duration variable set by the programmer that defines the number of times it has to be run. The pointer `fct` is changed as soon as this variable reaches zero.

Another important factor is the usefulness of the function `set_address`. Suppose we have the stride sequence example used at the beginning of this section, and suppose it represents the strides occurring from a certain base address. If this address changes all the time, for example from successive calls to the same function, and the stride sequence does not, we would have many different sequences in our graph. Being able to set the base address before the model considers the strides significantly reduces the number of needed nodes.

4 Experiments

To implement a prefetch mechanism, a way must be found to load before-hand the data into the cache. A simple solution is to insert a load into the assembly code before the data is actually needed. The drawback of such a solution is that the address must necessarily be valid. We are not allowed to load an address not situated in the memory space of the current program. Therefore we would have to check whether our strides do not let the prediction exit the possible address range. It is in general unconceivable for a low overhead to put up these tests and still get a speedup. The simplest solution is to use the prefetch instructions existing on most modern architectures.

On Itanium, the prefetch instruction is *lfetch*. However if there are too many outstanding prefetches, and if it provokes a page miss, then the prefetch is not executed.

All the programs presented here have been compiled with the optimization level "-O3" and run on an Itanium-2 processor. We used both compilers *GNU gcc* and *Intel icc*.

```
m1 = initialize();
set(m1,&stop,M_ARRCONS);
set(m1,&j,M_PROF);
set(m1,&k,M_ERRMAX);
```

Figure 2: Initialization code of the strucutre

The instrumented programs come from the different benchmarks *Spec*2000, *Pointer Intensive* and *Olden*.

4.1 `mcf` from the Spec2000 benchmarks

In what follows, we detail the instrumentation of the `mcf` program. This program can be accelerated at least at two distinct locations, so two data structures are used to monitor both stride sequences. Figure 2 shows the code used to initialize a structure. First, we call the `initialize` procedure that puts all the parameters to default values. Then it is possible to specify the parameters: the number of calls per phase, the depth of the construction, the number of consecutive misspredictions before a flush occurs, ...

On figure 3, we can see that only a few added function calls (in this case three) is necessary to monitor a certain sequence. For the sequence monitored by the structure `m2`, two calls are necessary since the pointer it is monitoring is updated at two different locations. Notice how the right function is called by using the pointer `fct` situated in the different Markov structures. The functions have two parameters:

1. The pointer to the Markov structure, where the graph and the pointer function can be found ;

```
...
while( arcin ) {
        tail = arcin->tail;

        m1->fct(m1, arcin->tail);

        if( tail->time + arcin->org_cost > latest ) {
                m2->fct(m2, (void *)tail->mark);

                arcin = (arc_t *)tail->mark;
                continue; }
        ...
        m2->fct(m2, (void *)tail->mark);
        arcin = (arc_t *)tail->mark; }
...
```

Figure 3: Inserting the `fct` function in the `mcf` program (in the function `price_out_impl`)

2. The pointer to the data to be monitored. Our functions transform the address sequence into a stride sequence.

Once the program is finished, we add a call to the `clear` function to free the memory used by our model. Figure 4 shows the acceleration of the program. It has been accelerated by 18% using the *icc* compiler. Only the single function `price_out_impl`, taking 50% of the whole execution time, has been optimized. Hence this function has actually been accelerated by 36%.

4.2 `treadd`, `equake`, `ks` and `mcf`

We present three other program experimentations: `treeadd` from the *Olden* benchmarks, `equake` from the *Spec2000* benchmarks and `ks` from the *Pointer Intensive* benchmarks for which one of the most costly functions of each program has been optimized (See table 3).

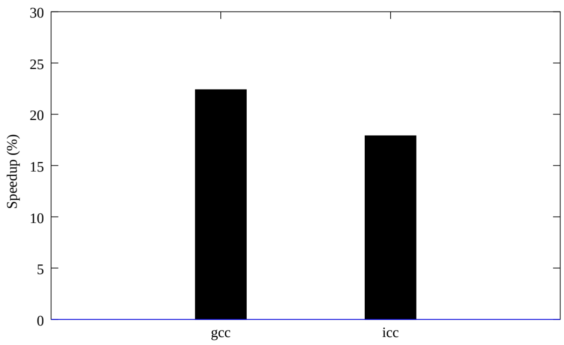

Figure 4: Speedup with the `mcf` program using a depth of 1 and distance of 4.

Program	File involved	Function involved	Input
treeadd	node.c	Treeadd	20 nodes and 1 processor
equake	quake.c	smvp	The reference input
ks	KS-2.c	FindMaxGpAndSwap	KL-5.in
mcf	implicit.c	price_out_impl	The reference input

Table 3: The functions yielding a lot of cache misses

The speedups achieved on these different programs can be seen on figure 5 and table 4. Notice that the given measures are resulting from the whole execution time of each program and not uniquely from the time of the optimized function. They were obtained with different parameters shown in the table. For example, if we look at the `treeadd` histograms, from left to right, they represent (1,1), (1,3) and (1,4) where the first value is the depth used by the Markovian model and the second value is the prefetch distance.

Program `treeadd` does not need a depth higher than 1 for speedup. Here our objective is to show the importance of choosing a good distance of prefetching. We changed this program so that it scans

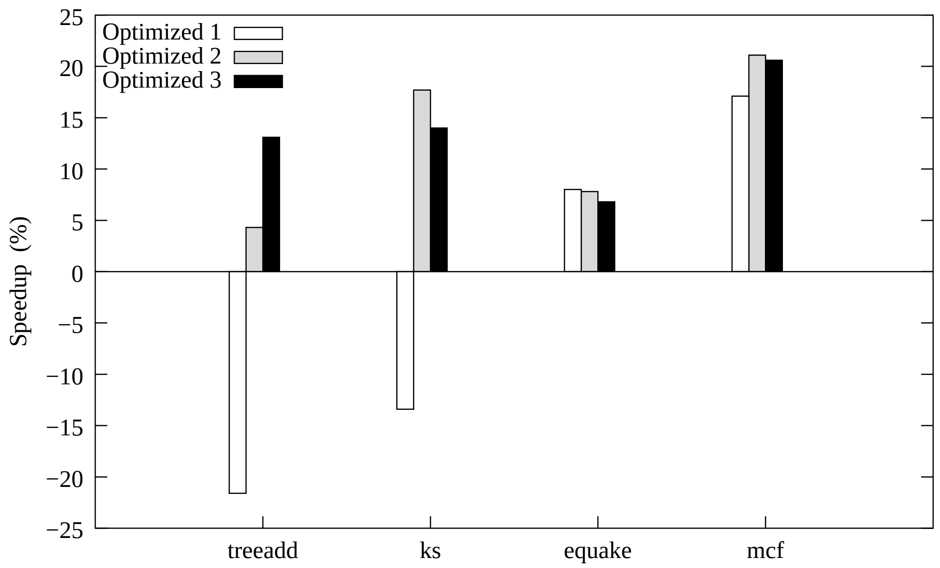

Figure 5: Speedup with `treeadd`, `equake`, `ks` and `mcf`. The meaning of the histograms is given by the table 4

the tree more than once (2000 times), the original execution time being too short. On the other hand, program `ks` does show a difference with a depth of at least two. Program `equake` works the best with a depth of 2 and a prefetch distance of 3.

For the program `mcf`, the performance decline when the prefetch distance is too high as it is shown by prefetching 9 strides ahead of time.

5 Conclusion

We have shown that a pure software dynamic optimizer is a realistic way of improving program behavior. Using some standard hardware mechanisms, it is possible to define a generic dynamic

Program	Depths	Distances
treeadd	1/1/1	1/3/4
ks	1/2/4	3/3/3
equake	2/2/2	1/3/5
mcf	1/1/1	2/4/9

Table 4: Different parameters used for the histogram in figure 5

process whose overhead stays sufficently low regarding the resulting speedup.

The presented memory strides Markov model provides significant improvements for data-intensive applications as shown in our experiments. Moreover it can easily be used for monitoring inter-procedural memory accesses. But it obviously cannot be an universal solution. It could also still be extended, for example by deleting the paths in the graph that are not followed very often, and so reducing the overhead induced by too many nodes in the Markov graph. However, since the room for manoeuvre is quite reduced, a smart balance between the optimization strategy and the induced overhead must be found.

The way we implemented our optimizer could also be advantageously changed for example by trying to avoid the necessary overhead of a function call with parameters.

We plan to explore other optimization strategies for prefetching, but also for other goals as dynamic data locality optimization or dynamic generation of cache hints. The overall objective is to build a global dynamic optimization framework where several strategies can be selected depending on the kind of monitored memory accesses.

Hence another challenging objective is to render as transparent as possible the usage of such a framework, by including program phase detection and classification processes guiding efficiently the locations and natures of the optimizations.

Dynamic optimization can never be as efficient as static optimization while handling static control and data structures. That is why we exclusively focus on variable-dependent control and memory accesses where dynamic optimization is undoubtedly a convenient answer.

ESODYP is available by request to the authors.

References

[1] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 52–61. ACM Press, 2001.

[2] J.-L. Baer and T.-F. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[4] K. Beyls and E. D'Hollander. Compile-time cache hint generation for epic architectures. In *Proceedings of the 2nd workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techniques*, November 2002.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st International Symposium on Code Generation and Optimization*, 2003.

[6] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209. ACM Press, 2002.

[7] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 62–73. IEEE Computer Society Press, 2002.

[8] M. Crochemore and M.-F. Sagot. *Handbook of Computational Chemistry*, chapter Motifs in sequences: localization and extraction. Marcel Dekker Inc., 2004.

[9] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: a new run-time control point. In *35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 257–268, December 2002.

[10] Intel. Hyper-threading technology. http://www.intel.com/technology/hyperthread/.

[11] Intel(R). *Itanium(R) 2 Processor Reference Manual for Software Development and Optimization*, chapter Optimal use of lfetch, pages 71–72. Intel Corporation, May 2004.

[12] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *IEEE Transactions on Computers, Vol. 48, NO. 2*, pages 121– 133, Febuary 1999.

[13] D. Kim, S. Liao, P. Wang, J. del Cuvillo, X. Tian, X. Zou, D. Yeung, M. Girkar, and J. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 27–38, 2004.

[14] J. Kim, K. V. Palem, and W.-F. Wong. A framework for data prefetching using off-line training of markovian predictors. In *20th International Conference on Computer Design (ICCD 2002)*, 2002.

[15] 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 Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003.

[16] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th annual international symposium on Computer architecture*, pages 40–51, 2001.

[17] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 115–126. ACM Press, 1998.

[18] H. Zhou, J. Flanagan, and T. M. Conte. Detecting global stride locality in value streams. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 324–335. ACM Press, 2003.