TDT4260 Computer Architecture Mini-Project: Development and Evaluation of Different Prefetchers Using M5 Simulator

Stefano Nichele, Angelo Spalluto

***Abstract*—In the last 50 years, the number of transistors in electronic circuits have increased following the Moore's Law and thus, performances of processors have also increased. On the other hand, performances of memories have not increased as much as microprocessors. In order to reduce this gap, other hardware and software mechanisms have been implemented. Since microprocessors can execute instructions faster than when the actual data will become available from physical memory, fast cache memories have been introduced to keep and deliver (when necessary) the needed data. A key component that helps the interaction between processor, cache and physical memory is the prefetcher. Prefetching means predicting which data will be needed by the next executed instructions and fetching it into the cache memory in a way that it will be available before it will be referenced. In this paper we will analyze two different types of L2 prefetching mechanisms suitable for different program structures, sequential and not-sequential, and we try to combine them together.**

***Index Terms*—computer architecture, prefetcher, cache**

I. Introduction

Transistors are semiconductor devices used to build integrated circuits and electronic components. Obviously, the performances of microprocessors are directly related to the hosted number of transistors. A direct consequence is that smaller are the transistors, higher is their density on a chip. Therefore, the microprocessors manufacturers started in the early '70s a miniaturization process. This trend was described by Intel co-founder Gordon Moore. He guessed that the growth of the number of transistors on integrated circuits would double every two years. Moore's Law [1] turned out to be particularly accurate until 2002 (it is still valid for FPGAs and multi-core processors) but physical constraints and the emergence of green computing [2] slowed down this process. Nevertheless, even if the size of memories has incremented, their speed has not improved enough. Memory latency has become the bottleneck in modern computers' performance. This situation is also referred as "*memory wall*" [3].

Last version: April 7th, 2011. This document is written as a partial fulfillment of the course TDT4260 – Computer Architecture.

Stefano Nichele is a PhD student at the Norwegian University of Science and Technology (nichele@idi.ntnu.no).

Angelo Spalluto is a MsC student at the Norwegian University of Science and Technology (spalluto@stud.ntnu.no).

Several mechanisms, such as memory hierarchies [4], Out-of-Order (OoO) execution, NUMA [4], increase of bandwidth utilization and CMPs [5], have been exploited to mitigate this phenomenon. The introduction of a memory hierarchy consists on the incorporation of small and fast cache memories between the processor and the RAM. This is shown in Figure 1.

Data that is frequently used can be placed in the cache memory which is physically close by the processor and therefore can deliver the required information faster. Level1 cache is usually small and placed on-chip, L2 is bigger than L1 and slightly slower. In CMP architectures, those two levels of cache are often private to each specific core. There may be a L3 cache (and even more levels) which can be shared among the cores and placed off-chip.

Now that the hardware architecture supports a technique to decrease the memory latency, it is required to implement a mechanism to load data from the memory onto the cache hierarchy, i.e., prefetching.

Prefetching is a speculative technique that aims to predict which data will be used in the future and fetches it into the cache memory before it will be required by the CPU. This can be done by detecting patterns in the program execution, but since it is just a mere prediction, the prefetcher can be wrong and thus pollute the cache and overload the bandwidth. The situation where the data referenced by the CPU is found in the cache is called "*hit*" and, on the other hand, if the data is not available in the cache and it has to be loaded from the main memory there is a "*miss*". In other words, an effective prefetching reduces the number of misses, exploiting cache properties such as spatial and temporal locality.

Definition 1: Spatial Locality
If a data is accessed, it is likely that other data in addresses nearby will be accessed in the future.

Definition 2: Temporal Locality
If a data is referenced by the CPU, it is possible that it will be used again in the future.

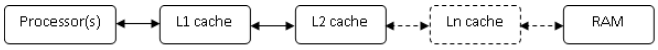

Fig. 1. Memory hierarchy example.

In chapter II a brief explanation of the most common prefetching algorithms is given. Chapter III describes the techniques that we have implemented and Chapter IV illustrates the used methodology. In Chapter V a discussion of the results is presented. Finally, Chapter VI concludes the paper.

II. Related Work

Several mechanisms, such as sequential prefetching [9], stride direct prefetching (SDP) [10], reference prediction tables (RPT) [11], program counter/delta correlation prefetching (PC/DC) [12], delta correlation prediction table (DCPT) [6] and adaptive prefetching [13], have been studied in the past.

Sequential prefetching exploits the spatial locality property, simply issuing the next block whenever a cache miss is detected. This can be improved adding a tag bit which indicates that the cache block was fetched by the prefecther. Unfortunately, not all programs access memory locations in a sequential way. In *SDP*, each time a program counter is encountered, it is possible to determine the difference between the two last referenced memory blocks and calculate the next block to fetch simply adding the last required address to the calculated delta. An improvement to SDP is *RPT*. The main idea is to save the PC and the relative referenced address (initial state). The second time the same PC is recalled, the delta is calculated (training state). The third time, a new delta is calculated and, if it matches with the previous, the prefetcher starts to operate (prefetching state). *PC/DC* uses a global history buffer (GHB) to store the chronology of each miss. The entry in the GHB is connected with the previous and the calculated deltas are stored in a separate table. *DCPT* stores, for each PC, a certain number of computed deltas and, if there is a match between the last two deltas and all the deltas that previously occurred, than the prefetcher calculates the next memory addresses to fetch based on the previous history. Finally, *adaptive prefetchers* try to adapt to the specific situation, being more aggressive and speculative when the prefetching is performing positively and being more conservative when the behavior is not as expected.

III. Implemented Prefetchers

In the beginning, in order to develop a level 2 cache prefetcher that has decent performances on average in every situation, it is necessary to study the behavior of each benchmark available in the “SPEC CPU2000” suite, which is used to carry out the tests. To do so, we have executed the given sequential prefetcher, to understand the sequencing degree of the memory locations accessed by the benchmark programs. Afterwards, we have modified the sequential prefetcher increasing the “prefetched window” (the number of subsequent prefetched memory blocks at each iteration) from 1 to 6 (the results are shown in Figure 4 and described in Section V). Keeping a fixed size window for the whole prefetching process is a strong restriction. It is possible to give freedom to the prefetcher to adopt the number of prefetched items (the window size) depending on the local accuracy of the previously prefetched elements.

In other words, if N subsequent memory blocks that have been previously prefetched are referenced by the CPU (all or up to a certain threshold), the next time the prefetcher will fetch N+1 memory blocks. If the local accuracy inside the defined window is lower than the threshold, the prefetcher will reduce the size of the window and select only N-1 elements.

With some benchmarking programs we noticed that, due to a low sequencing degree, the adaptive window prefetcher was not improving its performances. We decided to implement an algorithm based on prediction of referenced memory blocks with delta correlations.

Finally, we decided to combine both algorithms together in a unique prefetcher.

In the following paragraphs, the technical details of each implementation are described.

A. *Fixed Sequential Prefetching*

Even though the results of Fixed Sequential Prefetching are widely known in literature, we have decided to use this algorithm, in order to extrapolate the degree of sequencing for each benchmark. The strategy adopted in this prefetcher is quite straightforward and it uses only the concept of *block size* to predict the possible future addresses requests. The block size is the area that contains a specified number of contiguous pages, used to exploit spatial locality. The dimension of block size is tricky to calculate. In fact, if the value is too high, it affects cache performances and interconnection bandwidth. By contrary, if the size is too low, the sequential algorithm might issue less requests than those needed. In our algorithm, we have used the value proposed in the given example of the mini-project guide [7].

The used sequential algorithm works as follows: whenever a miss occurs, it prefetches the N blocks following the missing block, where *N* is the *degree of prefetching*. The number of blocks issued after a miss remains constant for the whole execution of the program.

Moreover, the algorithm checks if the address intended to issue is not already present inside the cache or in the Miss Information Status Holding Register (MSHR). This further control allows reducing the amount of transferred data, therefore also *interconnection bandwidth* and *cache pollution*.

B. *Sequential Aggressive Adaptive Window Prefetcher*

The aggressive sequential adaptive algorithm [13] is an improvement of the previous one and it also allows exploiting better the sequencing of each benchmark. The implementation of this approach requires more memory than the sequential version. The main benefit of this algorithm is due to a window

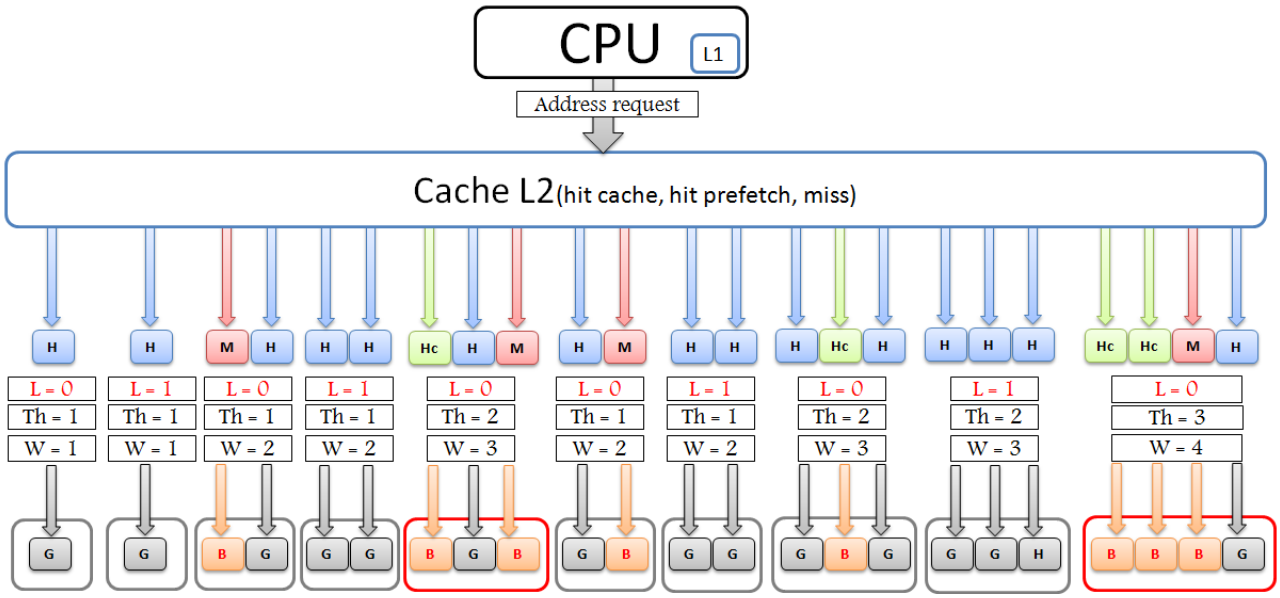

Fig. 2. Example of Sequential Aggressive Adaptive Window Prefetcher.

that is dynamically adjusted to the specific benchmark at runtime. The Adaptive prefetcher changes the degree of prefetching according to the previous value of the window. Hence, it increases the window only when it is sure that the next contiguous blocks might have a high probability to be used by CPU. In this way, the algorithm provides a different degree of prefetching while running the same benchmark.

In order to increase the reliability of the current window, the adaptive algorithm introduces the concepts of *threshold* and *lock window*. The algorithm uses these two values as constraints to respect before updating the value of its window.

When the algorithm receives a miss or hit, it performs prefetching for the following N blocks, where N is the value of dynamic window. Moreover, the algorithm sets the corresponding *tag bit* in the cache block for those requests issued by the prefetcher. After that, it remains in a *listening state* for the next N requests (might be hits or misses).

In this state, it counts how many hits have the tag bit set, hence, how many of them have been issued by the prefetcher (it discards those hits without *tag bit* set). The calculated value represents the *accuracy* of the previous window or, in other words, how many of N requests issued by prefetcher have been used by CPU. After that, the value of N for next prefetching might increase or decrease. To do that, the algorithm compares the accuracy value with a threshold. If the accuracy is greater or equal to the threshold, it means that the prefetching achieved a good result and it will work again with same window size or even more sequencing (bigger window). Otherwise, it means that the level of sequencing in the running code is decreasing (smaller window). There might be a problem when N continuously oscillates between two or more different window sizes. This can be solved using a conservative technique and keeping the same window width for more steps (L iterations, where L is the number of times that the window is locked and cannot be modified).

If after L times the window is still valid (accuracy is greater or equal than the threshold), the algorithm increases the width of the window. Otherwise, the first time that the accuracy is lower than the threshold, the window will be immediately decreased and the L count will restart.

Figure 2 shows an example of Aggressive Sequential Adaptive algorithm. The prefetcher receives the requests issued by CPU throughout L2 cache (it can be a hit prefetch, a miss or a hit cache). *Hit prefetch* is a situation when a hit occurs and the tag bit is set. This means that the address was issued before by the prefetcher (blue square). *Hit cache* means that the address is present inside L2 cache with a null tag bit (green square). The *miss* request indicates that an address is not found in L2 cache (red square). Figure 2 also shows for each request, the value of: current *window* (W), *threshold* (Th) and *lock flag* (L). The example uses a max value of L equal to two and Th=W-1 (when W=1, Th is also one).

Initially, the values of W and Th are equal to one. When the first hit occurs, the prefetcher issues only one request that turns out to be a good prefetching (grey square).

Since L<2, before increasing the window another step is required (with W=1). The first time with W=2, only one request turns out to be a good prefetching but this condition is still valid because Th=1 and it still satisfies the requirements (# of good prefetching >= threshold). The second time with W=2, the algorithm performs again successfully and it updates W=3 (window size is equal to 3). Unfortunately, at the next stage, only one request turns out to be a good prefeching (one of the received hit is not triggered by the prefetcher). Since the threshold is two, the condition is no longer satisfied and

the size of window is reduced (W=2). Thereafter, the algorithm works as before, restarting from W=2. The next stages show that the algorithm reaches window equal to three and four but, with W=4, the number of good prefetched addresses is not sufficient to satisfy the corresponding threshold (Th=3).

As shown in the example, the value used as threshold represents a good tuning to enhance the performances of the whole algorithm. The final value has been chosen after many tests.

It is important to remark that prefetching does not occur until all the elements inside the window have been checked (P elements = #hits + #misses). This solution is called aggressive sequential adaptive because it issues a prefetching right after the end of each window, either if the last item was a hit or a miss. Besides, two other less aggressive approaches have been studied.

The first approach, called *Miss-Adaptive* (M-Adaptive), issues prefetching only when (once the checks on the previous window have finished) the first miss occurs. Even in this case, P = #hits + #misses.

The second approach, called *Discard Miss-Adaptive* (DM-Adative), issues a prefetching immediately after the first miss occurs inside the window (P = H, where H are only hits).

Even if those two last approaches seem more reasonable, we decided to use the aggressive solution because the achieved results were better.

C. *Delta Correlation Prediction Tables (DCPT) Prefetcher*

Delta Correlation Prediction Table [6] is a technique that combines the main principles of RPT and PC/DC. It saves, for each program counter that produces a memory request, the deltas between subsequent requested memory blocks. With those deltas, it is possible to keep track of the pattern with which the program accesses memory. If a repetitive pattern is found, DCPT calculates the address of the memory block that most likely will be required at the next execution of the same instruction and therefore deliver it in advance. This mechanism is implemented using a table that stores the Program Counter (PC) of the executed instruction, the address of the last accessed memory block, the history of deltas in a circular buffer and a pointer to the last delta.

In literature, DCPT stores also the last prefetched address. In our implementation we have ignored this field in order to reduce the size of the table. In our opinion, the lack of this field is mitigated by the check of the presence of a certain memory block in cache before the actual prefetching (it may happen that the memory block is still in the buffer waiting to be loaded in the cache). When the instruction with the same PC is executed again, it is possible to travel backwards in the deltas table, searching for equal delta patterns that previously occurred. Tuning and adjustments are required in order to find the balance between size of the table and performances. Important parameters are the number of stored deltas and the number of stored PCs (an increase in one is reflected as a decrease in the other).

Sometimes, the deltas are not perfectly matching but the required addresses are very close by. An improvement that can be introduced if no pattern repetitions are found is *Partial Matching*. The aim of PM is to reduce the spatial distance of the delta sequence and identify similar patterns. This is done by masking the less significant bits of the deltas and comparing only the most significant ones.

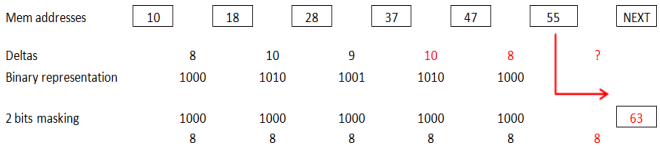

Fig. 3. Example of DCPT with partial matching.

In Figure 3, without bit masking the two last deltas 10 and 8 are not found in the delta stream and the prefetcher is not able to predict which memory block will be useful in the future. It is evident that, even if there is no perfect matching, all the deltas have close values. It may be beneficial to prefetch a memory block calculated with an enough accurate delta. Hiding the least two significant bits of the delta stream, it is possible to find a "*partial*" matching of the last two masked deltas (8 and 8) with some previous values and thus to calculate the following element to deliver.

D. *Adaptive Delta Correlation Prediction Table (WA-DCPT and SA-DCPT) Prefetcher*

In this section two different approaches to combine Sequential Aggressive adaptive Prefetcher and DCPT have been proposed: *Window Adaptive-DCPT* (WA-DCPT) and *Switch Adaptive-DCPT* (SA-DCPT).

WA-DCPT introduces a different window for each PC stored in the Prediction Table. When DCPT needs to issue a prefetching for a specific PC, it delivers also all the subsequent blocks, according to its window size. This solution, compared with DCPT, has a more memory demanding data structure because it also needs to store the value of windows associated for each PC. This value is extremely important because, when the algorithm uses again the same PC, it needs to utilize again the same window size used before. Thus, each time the prefetcher identifies a new PC, it saves the previous window and it loads the new values of window and threshold. Yet, the rules to update the window are the same explained in adaptive sequential algorithm.

SA-DCPT uses a different strategy. It tries to adapt the best algorithm for each benchmark, exploiting the benefits of both sequential adaptive and DCPT. Every time the prefetcher receives a request by L2 cache, it checks the current value of window and if it is less than a *specific threshold*, it employs DCPT, otherwise it uses the aggressive adaptive algorithm. The value of window used to switch from an algorithm to

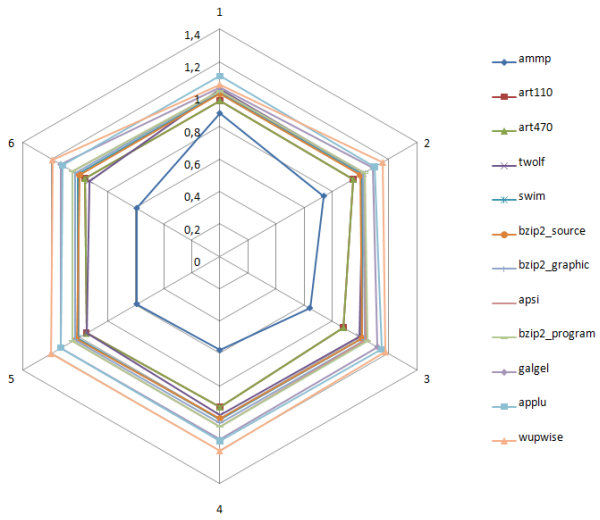

Fig. 4. Comparison of the behavior of different benchmarks. The size of the prefetched static window has been varied from 1 to 6. Each edge of the hexagon represents a window size and the height of the hexagon represents the correspondent speedup.

another represents the main issue of this algorithm. From our results, we have observed that the best window parameter for switching among benchmarks is W=4.

IV. Methodology

To test the performances of the different prefetchers, we have run the SPEC CPU2000 benchmarking suite under M5 simulator. The system architecture is an OoO CPU with Alpha 21264 microprocessor, 32kB L1 cache (without prefetching) and 1MB L2 cache [7].

The L2 cache prefetcher is notified every time there is a hit or a miss in L2 cache. The size of the cache block is 64bytes and the maximum number of pending prefetch requests is 100 (MAX_QUEUE_SIZE). The simulated CPU clock runs at 2GHz while the memory bus has a frequency of 400MHz. The size of the physical memory is 256MB.

The prefetching algorithm is mainly composed by three functions: *prefetch_init*, *prefetch_access* and *prefetch_complete*. Modifying those functions it is possible to change the prefetcher's behavior. The first function (prefetch_init) is called before the first access to memory and it contains all the initializations of the declared data structures. The second function (prefetch_access) is the main function which is called every time the prefetcher is informed that the CPU has accessed the L2 cache through the L1 cache (hit or miss). Inside this function the actual prefetch request is executed through a call to the function *issue_prefetch*, which queues a specific address into a *buffer*. Finally, the last function (prefetch_complete) informs the prefetcher that a previously queued request for a memory address has been accomplished and the actual memory block has arrived in the L2 cache.

V. Discussion of the Results

The results presented in this paragraph are obtained running simulations on Virtual Machines with Linux Operating System. The package is described in [7].

A. Comparison of Sequential Prefetching algorithms with fixed size windows

The radar graph in Figure 4 shows a comparison of the behavior for each benchmark. Each corner of the hexagon illustrates the speedup achieved by a static sequential algorithm with fixed size windows prefetching.

It is notable that benchmarks such as *wupwise*, *applu* and *galgel* have considerable results with this algorithm. Moreover, the performances of those benchmarks slightly increase with the size of the window. In other benchmarks (*swim*, *bzip2_source*, *bzip2_graphic*, *apsi*, *bzip2_program*) the performances are steadily around the same value.

Unlike the previous benchmarks, *ammp* does not perform as expected using a sequential prefetcher with a fixed size window and it even decreases the performances with a lager window. The obtained performances are also slightly lower with *art110*, *art470* and *twolf*.

We think that to obtain good results in the whole suite, we need to develop a hybrid strategy that uses both sequential and not-sequential algorithms.

B. Adaptive Window

The graph in Figure 5a compares the behavior of different aggressive adaptive sequential algorithms (*ADAPTIVE-MaxWinN*) using different maximum windows' size (*N*). Additionally, the best results for the sequential algorithm are also presented (*BEST-SEQUENTIAL*). Among the different adaptive prefetchers, on average, the best performances are achieved using an adaptive algorithm with a maximum size of 12. According to our tests, using a window greater than this value, the performances are steady. The results of *ammp*, *art110* and *art470* confirm the trend shown in section A (low sequencing). In fact, an increase of the max size of window does not improve the speedup.

Figure 5b introduces the results of other two variants of Adaptive Sequential: *M-Adaptive* and *DM-Adaptive*. The achieved results are not better than the Aggressive Adaptive (with MaxWin=12). As expected, these two algorithms produce less "*misses*" and "*prefetchers issued*". In fact, since the prefetcher issues requests only when a miss occurs (the first miss inside the window for DM-Adaptive, the first miss after the window in M-Adaptive), it reduces the whole amount of misses.

C. Adaptive vs DCPT vs DCPT-P

The graph in Figure 2c compares the best adaptive algorithm (max window size of 12) with DCPT and DCPT-P.

The chart illustrates that for *ammp* the DCPT-P prefetcher outperforms almost twice better than the adaptive. Furthermore, for all other benchmarks the performances are similar or oscillating inside a limited interval. As expected, the results of DCPT-P are slightly better than DCPT. This result is also present in literature [14].

In order to use a data structure smaller than 8Kb, we have used a table composed by 16 deltas and 97 PCs. Furthermore, for DCPT-P we have used a masking of 8 bits. According to [14], masking more that 12 bits does not produce any *Speedup* enhancement. In our tests, we have not observed differences between a 8 bit or 12 bit masking.

D. Adaptive Delta Correlation Prediction Table (WA-DCPT and SA-DCPT)

Before comparing WA-DCPT and SA-DCPT, we have tried to perform some tunings in order to achieve better results.

In WA-DCPT, the number of PCs and deltas has been varied (all configurations are smaller than 8KB) as follows: *7x178*, *10x140* and *14x110*. Respectively, the first value is the number of deltas and the second value is the number of PCs. Best results have been achieved using 14 deltas, as illustrated in Figure 5d. This tuning is compliant with the results presented in [15].

In SA-DCPT we tried to vary the triggering event for the switching between the combined algorithms. If the window size is less than a specific threshold, DCPT is used. Otherwise Aggressive Adaptive is utilized. Threshold has been tested with values from 1 to 4. Unlike WA-DCPT, the chosen configuration is 16x97 (the data structure is different). We noticed that the best speedup is achieved with a window threshold equal to 4. The results are presented in Figure 5e. For greater values, SA-DCPT behaves as a normal DCPT.

Considering only the best results (SA-DCPT_16x97-W4 and WA-DCPT_14x110), both algorithms have the same behavior for all benchmarks except *twolf* and *ammp* (negligible difference).

E. Comparison of developed prefetchers

The graph in Figure 5f is a summary of the performances achieved by all developed algorithms, such as *Fixed Size Sequential Prefetcher* (BEST_SEQUENTIAL), *Aggressive Adaptive Window Sequential Prefetcher* (ADAPTIVE-MaxWin12), *Delta Correlation Prediction Table with Partial Matching* (DCPT-P_16x97) and *Switch Adaptive DCPT* (SA-DCPT_16x97-W4). Since the results of SA-DCPT and WA-DCPT are close by, we decided to use SA-DCPT in the following comparisons, also because it performs slightly better with *ammp* benchmark.

The results in Figure 5f show that the DCPT-P obtains the best performances. It is important to highlight that, right after DCPT-P, the second best is SA-DCPT. In fact, it performs better than the other algorithms except for *swim* benchmark. SA-DCPT is a good compromise when there is a situation with both sequential and not-sequential executions in the running programs at the same time.

F. Comparison overview

The graph in Figure 5g illustrates the behavior of all developed algorithms together with reference prefetchers available in literature. An important remark is that our implementation of DCPT-P outperforms the reference DCPT-P. We believe that this result is achieved because we are using a different implementation and data structure.

G. Analysis of Developed Prefetchers' Coverage

Coverage is an important statistic when evaluating prefetchers. It represents how many cache misses that could have occurred without prefetching are avoided. Thus, a coverage value close to 1 means a lower number of cache misses.

In Figure 5h, an analysis of the coverage for the developed prefetcher is shown. Benchmarks with low sequencing (ammp, art110 and art470) have a higher coverage with DCPT-P. On the other hand, benchmarks with high sequencing have better coverage with SA-DPCT (except applu).

Is it important to emphasize that a *coverage* value close to one does not implicitly correspond to a high performance in terms of *speedup*. In fact, in Figure 5f, *swim* achieves the worst speedup with SA-DCPT and, on the other hand in Figure 5h, the same algorithm obtains the best coverage. Our guess is that if the algorithm spends too much time discovering the next elements to prefetch, as consequence it might increase the execution time, therefore lowering its speedup.

VI. CONCLUSION

The major contribution of this paper is the development of three new types of prefetching, studied with different tunings. The first new class is an evolution of the sequential prefetching with an adaptive window. Several variants have been implemented, such as DM-Adaptive and M-Adaptive. The second group includes a DCPT-based prefetcher with partial matching and a data structure with different design. Finally, we proposed a combination of those two algorithms, integrating an adaptive strategy together with a delta correlation (WA-DCPT and SA-DCPT).

Even though DCPT-P outperforms all the other prefetching algorithms, we believe that adopting solutions such as WA-DCPT or SA-DCPT represents a good compromise in those cases when the type of benchmark is unknown. Our initial expectations about these two algorithms were higher than the actual results. In fact, we thought that embedding both algorithms it could be possible to achieve top performances in every case, either sequential or not-sequential code.

On overall, the mini-project was a very positive and interesting experience, especially to gain detailed knowledge in the field of computer architecture.

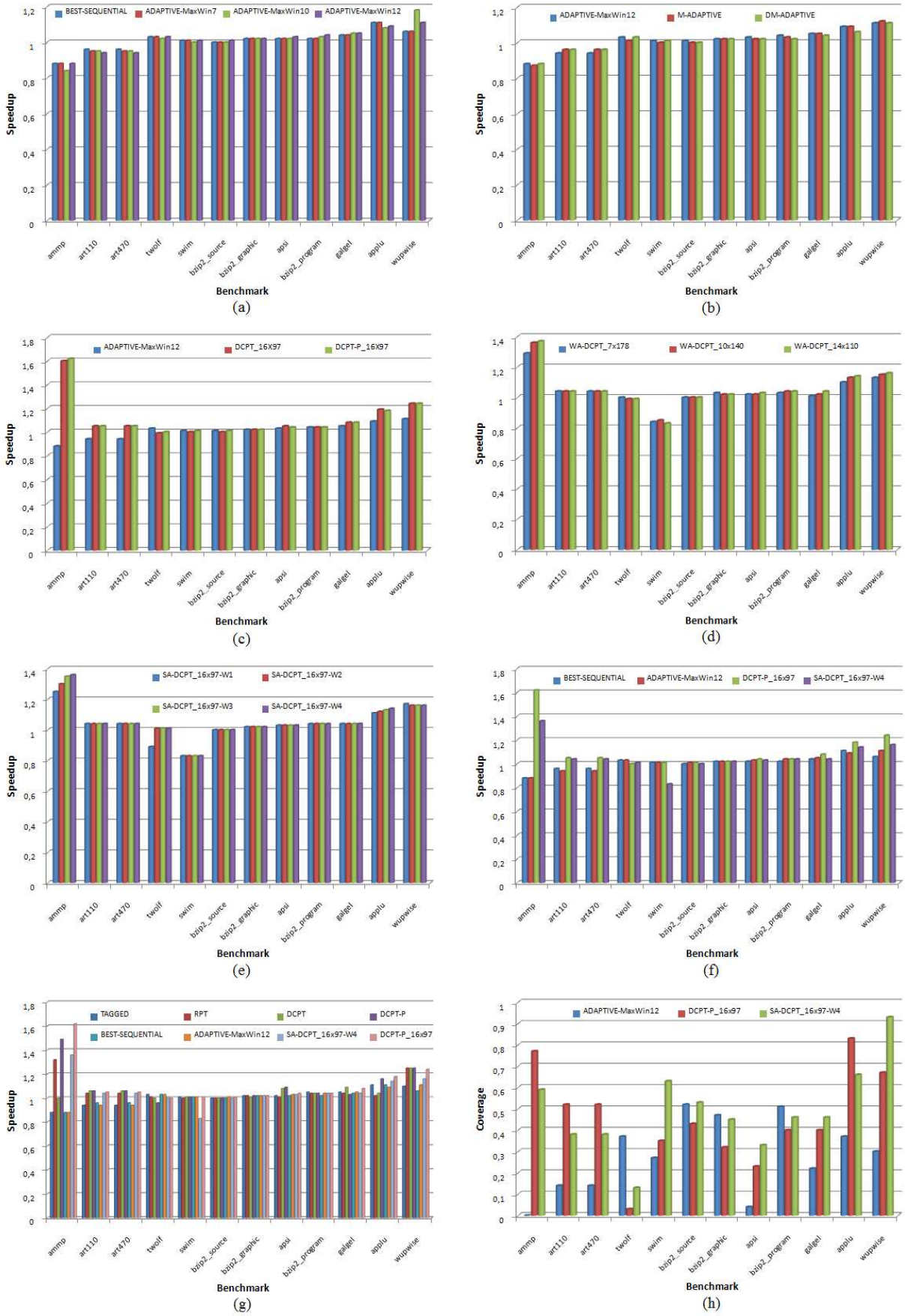

Fig. 5. (a), (b), (c), (d), (e), (f), (g): Comparison of speedup of benchmarks using different prefetching algorithms, either those developed by us and those described in literature. (h): Comparison of coverage of benchmarks using different prefetching algorithms developed by us.

REFERENCES

[1] G. E. Moore, Cramming more Components onto Integrated Circuits, Electronics, 38(8), April 9, 1965.

[2] A. Iordan, Introduction to Green Computing and Asymmetric multicore processors, TDT4260 Computer Architecture Lecture Notes, NTNU, March 25, 2011

[3] W.A. Wulf and S.A. McKee, Hitting the Memory Wall: Implications of the Obvious, Computer Architecture News, vol. 23, no. 1, Mar. 1995, pp. 20–24

[4] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, O. O. Storaasli, State-of-the-art in heterogeneous computing, Sci. Program., Vol. 18 (January 2010), pp. 1-33.

[5] M. Jahre, Managing Shared Resources in Chip Multiprocessor Memory Systems.: NTNU 2010 (ISBN 978-82-471-2287-7) 238 s. Doktoravhandlinger ved NTNU (159)

[6] M. Grannaes, Reducing Memory Latency by Improving Resource Utilization.: NTNU 2010 (ISBN 978-82-471-2177-8) 242 s. Doktoravhandlinger ved NTNU (106)

[7] A. C. Iordan, TDT4260 Computer Architecture Mini-Project Guidelines, January 10, 2011

[8] A. C. Iordan, M5 Simulator System.TDT4260 Computer Architecture. User Documentation, February 4, 2011

[9] A. J. Smith, Cache memories, ACM Comput. Surv., vol. 14, no. 3, pp. 473–530, 1982

[10] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture, pages 102-110, 1992

[11] T.-F. Chen and J.-L. Baer, Effective hardware-based data prefetching for high-performance processors, Computers, IEEE Transactions on, vol. 44, pp. 609–623, May 1995

[12] K. J. Nesbit and J. E. Smith, Data cache prefetching using a global history buffer, High-Performance Computer Architecture, International Symposium on, vol. 0, p. 96, 2004

[13] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In Parallel Processing, 1993. ICPP 1993. International Conference on, volume 1, pages 56-63, Aug. 1993.

[14] M. Grannaes, M. Jahre and L. Natvig. Multi-level Hardware Prefetching Using Low Complexity Delta Correlating Prediction Tables with Partial Matching. High Performance Embedded Architectures and Compilers LNCS, 2010, Volume 5952/2010, 247-261.

[15] M. Grannaes, M. Jahre and L. Natvig. Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables. In Data Prefetching Championships (2009)

Stefano Nichele is a PhD student at the Norwegian University of Science and Technology, under the supervision of Prof. Gunnar Tufte. He works with Bio-Inspired Architectures and Unconventional Computation. He obtained his MSc degree in Computer Science at Insubria University – Italy.

Angelo Spalluto is a MSc student at the Norwegian University of Science and Technology. He obtained his BSc at Politecnico di Torino – Italy – under the supervision of Prof. Silvia Chiusano.