A Table-Based Application-Specific Prefetch Engine for Object-Oriented Embedded Systems

Saahin Hessabi, Mehdi Modarressi, Maziar Goudarzi, Hani Javanhemmat
Computer Engineering Department
Sharif University of Technology
Tehran, Iran.
hessabi@sharif.edu, modarressi@ce.sharif.edu, goudarzi@sharif.edu, javan@ce.sharif.edu

***Abstract*—A table-based application-specific data prefetching mechanism is presented in this paper. This mechanism is proposed to improve the performance of the application specific instruction-set processors (ASIP) we develop customized to an object-oriented application. In this approach, we divide the data accesses of a class method into two conditional and unconditional parts. We supply the prefetch engine with the static information about each part to prefetch all data fields of an object required by a class method when the class method is invoked. Effective management of memory access patterns by dividing them based on the method to which they belong and storing the access information of nested loops using a simple structure are the merits of the proposed mechanism. In addition, by adding a prefetch flag to cache blocks, we eliminate a large number of prefetch related tag comparisons. The results show that the proposed mechanism reduces the cache miss ratio and prefetch related tag comparisons on average by 66% and 21% ,respectively.**

I. INTRODUCTION

Data prefetching is a basic technique for enhancing cache performance [1]. Although considerable research has been concentrated on prefetching techniques and many prefetching methods have been developed, since the memory-processor speed gap continues to increase, there is continuing demand for improving prefetching methods.

In designing a prefetching mechanism, it is important to pay attention to its impact on processor energy consumption. Cache memories may consume up to 50% of processor energy [2], [3]. Moreover, using some hardware prefetching mechanism will increase the memory system energy consumption by 30% [4]. However, in embedded systems where the hardware and the running application are specific and remain unchanged during system life time, some system attributes can be used to improve the prefetching performance and power consumption.

The application-specific processors we use in this research are introduced in Section 2. These processors are specifically designed to suit object-oriented applications and are synthesized from an object-oriented high-level specification using the ODYSSEY synthesis tool [5]. Class methods of the object-oriented specification are either implemented in software, as software routines, or in hardware, as hardware functional units. As a result, the processor is composed of a processor core, which executes the software routines, along with a number of hardware functional units.

We have already presented an application-specific data prefetching mechanism for these processors [6]. In the proposed mechanism, the cache controller prefetches all data fields of an object which are unconditionally accessed by a class method, when the class method is invoked. This approach adapts the prefetching mechanism to the running application. The simulation results of this mechanism using some object-oriented benchmarks show that on average, this method reduces the miss ratio by 70%.

In this paper, we propose a table-based implementation for that prefetching mechanism. We also customize this table-based structure for stride-prefetching in array-based accesses. Stride prefetching detects sequences of accesses that differ by a constant value and prefetches the addresses that continue the stride pattern. These methods often apply a table to store the most recent stride information [7] [8] [9].

Our approach divides the class method data accesses into two conditional and unconditional parts and takes advantage of static information about both parts to prefetch the required data items. A number of mechanisms such as the one presented in [4] apply compiler supplied static information to improve the performance and reduce the prefetching energy consumption. [4] uses a combined stride and pointer prefetching and supplies it with some static information such as stride information and the suitable prefetching method (stride or pointer) for each load/store instruction. It reduces the prefetch related energy overhead by 40%. Some other software-hardware cooperative prefetching approaches such as Data Prefetch Controller in [10], prefetching array in [11], User-Level Memory Thread in [12], Group Prefetching in [13], and Guided Region Prefetching in [14] apply the information about the running application to improve the prefetching performance. The prefetch structure we propose in this paper is simpler than the structures used in other software-hardware cooperative mechanisms while we can effectively detect the access patterns generated by class methods and prefetch the required data items before the actual request for them.

Although prefetching improves the cache performance, it increases the cache energy consumption. The main source of the prefetch-related energy overhead (especially in set-associative caches) is the cache lookups related to prefetching.

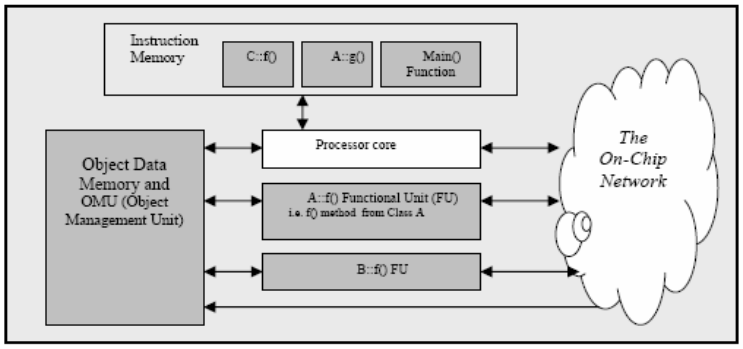

Figure 1. The internal architecture of our ASIPs

In set-associative caches, all the ways in a set are accessed in parallel although at most only one way contains the desired data. Since prefetch engines search the cache for a data before prefetching it, prefetching results in a large number of parallel tag comparisons and increases the cache power overhead. To alleviate this effect, we can add a small direct-mapped cache between the processor and the L1 cache to keep the prefetched data items. "Filter cache" [15] proposes a similar approach which stores the most recently used data items of the cache (but not the prefetched data) in a small direct-mapped cache. However, in order to eliminate moving data items between L1 cache and the prefetch cache, we propose an alternative way by adding a prefetch flag to every cache block in order to specify the recently prefetched (or hit during prefetch lookup) cache blocks. During the cache lookup, the cache controller first accesses the way specified by the prefetch flag (if any) and then searches the other ways if the first access results in a miss. This method is similar to the way-prediction methods proposed for power reduction in set-associative cache memories [16] [17] [18]. However, since we have associated it with prefetched data items, we can achieve higher hit ratios on the first phase accesses. In following sections, we will explain the details of our mechanism.

The structure of the paper is as follows. In Section 2, we introduce our embedded system architecture and present our prefetching mechanism. Section 3 contains the experimental results and finally, Section 4 concludes this paper.

II. DATA PREFETCHING MECHANISM

A. Embedded Processor Architecture

The embedded system architecture that we follow in this research is depicted in Figure 1 [19]. The system is a Network-on-Chip (NoC) architecture that consists of a processor core along with a set of hardware functional units (FU). The architecture is specifically designed to suit object-oriented (OO) applications. A typical OO application defines a library of classes, instantiates objects of those classes, and invokes methods of those objects. Our implementation approach for each of these three major components of an OO application is described below. For presentational purposes, we follow the C++ syntax in describing each component.

• Class library: Each class consists of variable declarations and method definitions. Variable declarations are compile-time information and do not require a corresponding component in implementation. Methods of the OO class library are either implemented in software (e.g. *A::g()* and *C::f()* in the "Instruction Memory" box in Figure 1) or in hardware (e.g. *A::f()* and *B::f()* FUs below the "Processor core" box). The execution of a hardware-implemented method takes multiple clock cycles and the method often follows the memory access pattern of its high-level description. However, hardware-implemented methods are several times faster than their software-implemented equivalents.

• Object instantiations: these are specified in the *main()* function. A memory portion should be reserved for each instantiated object to store the values of its data items. This memory portion is allocated in a data memory (the gray box at the left-hand side of Figure 1, called OMU: Object Management Unit) that is accessible to the processor core as well as all FUs.

• Method invocations**:** the sequence of method invocations is specified in the *main()* function of the application. The executable code of this function comprises another part of the instruction memory (see Figure 1).The processor core starts by reading the instructions specified in the *main()* function of the application. Whenever a method call instruction is read, the corresponding implementation is resolved and invoked. This may result in calling a software routine (e.g. *C::f()* in Figure 1) or activating an FU (e.g. *A::f()*). Each method implementation (be it in hardware or software) can also call other methods.

Since methods may be implemented in hardware as well as in software, new techniques are required to efficiently dispatch method-calls among method implementations. To achieve this goal, we view each method call as a network packet. Each method call is identified by a called method, a called object and the parameters of the call. Therefore, each packet includes three fields named *mid*, *oid* and *params* to respectively represent the called method number, the called object number and the call parameters. It also contains the *mid* of the caller method as the destination of return values. The numbers allocated to methods and objects are assigned such that routing the packet on the on-chip network corresponds to dispatching the call to the appropriate called method irrespective of the caller and called methods being in software or hardware [19]. If the invoked method is implemented in hardware as a functional

unit, the FU extracts the input parameters from the packet and starts execution. If the method is implemented in software, the processor core gets the packet and executes the method. The invoked method may also need to call other methods. During execution, the method may need to access some data fields of its corresponding object. The method requests the data by sending the *oid* and *offset* to the OMU. The OMU maps the *oid* and *offset* to a physical address and sends the corresponding data to the method.

When a method execution completes, the called method sends another packet over the same on-chip network to the caller method containing some fields such as return values. More details about this architecture can be found in [5], [19] and [20].

B. The Proposed Data Prefetching Mechanism

In previous section, we introduced the architecture of the object-oriented ASIPs. The OMU in our ASIP architecture also contains a data cache and controls it to accelerate data accesses. The cache controller monitors the network and extracts the called method and called object identifiers upon a method invocation. Thus, the cache controller can be aware of the currently called method and also aware of the object on which the method should work. It is also aware of the other part of the data addresses requested by a method (i.e. the data field offsets) by keeping them in a prefetch table. These data field offsets can be obtained by a static analysis of method codes and also profiling method codes using some reasonable benchmarks. Owing to the fact that class methods may be called on different objects, but the unconditionally accessed data fields of the called object are the same for all invocations, the cache controller being aware of calling a method on an object, can prefetch the unconditionally-accessed data fields of the called object to the cache. In our previous work, we simulated this approach on some object-oriented benchmarks and showed that it can reduce the miss ratio by about 70% [6].

Since array-based structures are the most common data structures in embedded systems, especially in image and voice processing and wireless applications [21], we focus on the stride prefetching [7]. Nevertheless, our mechanism is also able to prefetch scalar data items.

We divide the data accesses of a class method into two unconditional and conditional accesses. The unconditional accesses happen in the same way every time a method is invoked. The conditional accesses (such as data accesses in the body of an *if-then-else* statement or in a loop that accesses different elements of an array based on a variable value) take place based on the input parameters and the run-time behavior of the program. Figure 2 illustrates an example method code. In this figure, the first and the fourth loops are the unconditional access parts of method *f()*. On the other hand, since the second and the third loops access different elements of the arrays *b* and *d* based on the input parameter of the method, these loops form the conditional access parts of the method. Arrays *a, b, c* and *d* are the data members of class *cls* and *N* is a constant value. For the unconditionally accessed data fields, we use a table to keep their offsets. The table keeps the starting and stopping offsets and the stride of these access patterns (see below).

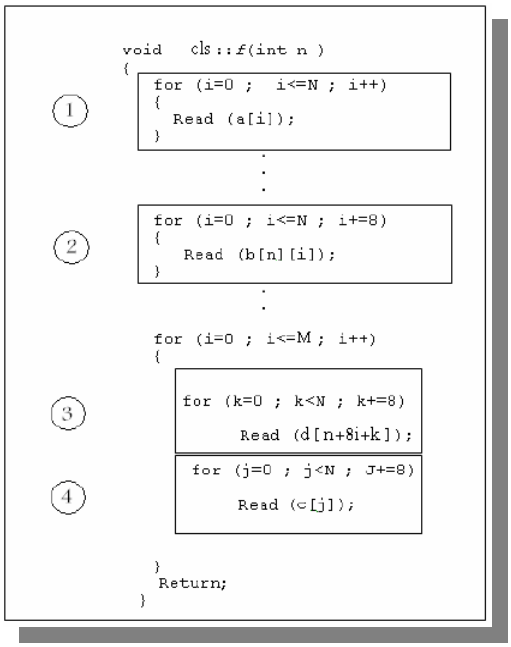

Figure 2. Pseudo code of a class method. The first and fourth loops generate unconditional access patterns while the second and third loops generate conditional access patterns

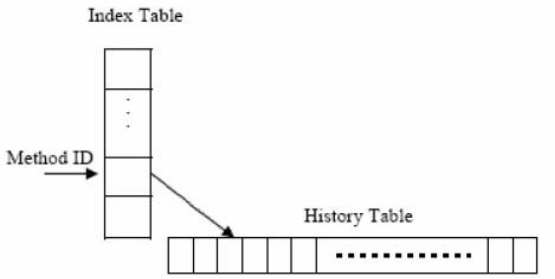

Figure 3. The structure of the proposed prefetch table

T 2 bits	S 1 bit	C 1 bit	Other Control bits	Start	Stride	Degree	Dyna.

Figure 4. An entry of the History Table

In the conditional data accesses in Figure 2, the start offset of the accesses depends on run-time conditions. However, even these accesses may still follow a constant stride. To support this type of conditional access patterns (with constant stride), we store the expected stride of them in prefetch table. The structure of our prefetch table is depicted in Figure 3. This structure has an *Index Table* which is indexed by the called method identifier (*mid*). The entries in the *Index Table* contain pointers into another table, *History Table*, which holds the memory access information of class methods.

Each entry of the *History Table* depicted in Figure 4 has some fields. The *Dyna* field is a dynamic field and stores the run-time information of the access patterns, while the other fields contain the static information provided before synthesis.

The *T* field determines the type of an entry. If *T=0*, the entry is used to prefetch a scalar data and holds a single offset in the *Start* field. If *T=1*, the *Start* field is the start offset of an unconditional access pattern. In this case, the *Stride*, *Degree* and *Dyna* fields keep the stride, the prefetch degree and the last prefetched offset of this unconditional access pattern, respectively. If *T*=2, the *Stride* field holds the expected stride for a conditional access pattern; The other fields are empty in this case. If *T*=3, the entry contains a pointer to another entry of the *History Table*. Accessing a pointer entry, if the *Dyna* field of the entry is greater than 0, the cache controller decreases the *Dyna* field by one and initiates prefetching according to the entry pointed by the *Start* field of the current entry. Here, the *Dyna* field acts as a loop counter and we use the *Degree* field to keep its initial value. The pointer entries are useful for prefetching the data items in nested loops. By using the pointer entries, we can simulate the nested loops and exactly follow their access patterns. By exactly following the access patterns of the methods, we can eliminate (or reduce) the cache pollution problem. The number of iterations of a loop generated by a pointer entry is stored in its *Degree* field and equals to the number of iterations of the corresponding loop in the application. The use of pointer entries enables us to store the access patterns of nested loops using a very simple structure. This is a major advantage of this table structure.

If the *C* field is set, the current and the next entries should be processed (and their addresses be prefetched) simultaneously. This is useful for handling the loops that access more than one array. In addition to the mentioned fields, the *S* field determines the end of the *History Table* entries related to a method.

As an example, five *History Table* entries are used to keep the access information of the loops in the pseudo code depicted in Figure 2. Upon invocation of the method *f()*To simulate the loop encircling the loops 3 and 4, we insert a pointer entry after the entry corresponding to loop 4. This pointer entry generates a loop through the entries related to the loops 3 and 4 and lets them prefetch *N* elements of their corresponding array at each iteration.

Unconditional data accesses can be easily recognized by their starting and stopping addresses; i.e., detecting the requests for starting and stopping addresses of an unconditionally accessed part determines the time when we should prefetch the unconditionally accessed data items according to the table. The cache controller uses the expected stride of a conditional access pattern to detect its start and uses either the first address of the next pattern (if the next pattern is an unconditional pattern) or the expected stride for the next pattern (if the next pattern is conditional) to detect its stop.

Since we focus on the data accesses of a single class method (and not the entire program) and each method contains a limited number of loops (and thus access patterns), we can often effectively manage data access patterns and divide them into conditional and unconditional parts and store the access information in the table. Moreover, by dividing data access patterns based on the methods to which they belong, detecting the access patterns (in order to initialize and stop prefetching) can be done more precisely at run-time.

Although the table cannot store the information of all access patterns, we can encode and store the access information of the access patterns generated by most of the common loop configurations. For more complex loops one can apply some algorithms to convert them into simpler patterns [21], [22]. Moreover, the irregular access patterns (for example the access patterns with variable stride) can be stored using the scalar entries in the table.

C. *Reducing Prefetch Related Tag Operations*

As mentioned before, accessing *N* data arrays and *N* tag arrays per cache access (where *N* is the associativity degree of the cache) is the main source of power consumption in set-associative cache memories. Since prefetch engines search the cache for a data before prefetching it, using a prefetch mechanism increases the number of tag comparisons, and thus the power consumption of the cache.

As an approach to reducing the prefetch related tag comparisons, one can add a small direct-mapped prefetch cache between the CPU and the L1 cache to keep the prefetched data items. This is similar to the filter cache idea [15] which keeps the most recently used data in such a direct-mapped cache.

However, to eliminate moving cache blocks between the L1 and the prefetch cache, we implement this prefetch cache within the L1 cache by using a prefetch flag for each way within a set. When we prefetch a block to the cache or when the lookup operation before prefetching hits the cache, we specify the way which contains the block by setting its prefetch flag. During cache lookup operation within a cache set, the cache controller first searches the way specified by the prefetch flag and then searches the other ways of the set if the first search does not hit the cache. The cache controller resets the prefetch flag, if the way specified by the prefetch flag does not contain the requested data. In addition, upon prefetching a new data into a set, we reset the prefetch flag of other ways within the set.

Like other way-prediction approaches, this approach leads to a two phase access and can potentially increase the cache access time. However, as we will see in the next section, since we only use the prefetch flag for the recently prefetched data which are very likely to be accessed after prefetching, most of the requests hit the cache at the first phase. As a result, this method effectively reduces the number of prefetch related tag operations while does not have a big effect on the cache access time.

Unlike most way-prediction approaches, however, in this approach, we only apply two phase access in the sets whose at least one prefetch flag is set. The cache performs the traditional parallel tag operations if no prefetch flag in the accessed set is raised.

In the next section we evaluate the proposed methods on some object-oriented ASIPs synthesized for real world applications.

TABLE I. THE CHARACTRICTICS OF THE BENCHMARKS

Benchmark	Input Parameter	No. of Memory References	No. of class methods(Functional Units)
JPEG Encoder	A 320×240 bitmap picture	5,733,229	17
JPEG Decoder	A 320×240 jpeg picture	4,632,957	40
MPEG Decoder	A 128x128x24b ppm file	4,927,940	97
V44	Two 32 kb text files	14,219,719	24
Sound Recorder	A random sequence of recording and playing .wav files	8,203,047	27

TABLE II. THE MISS RATIO VS. THE CACHE SIZE FOR DIFFERENT PREFETCHING SCHEMES

Cache Size	8 K			4 K			2 K		
Benchmark	*No Prefetching*	*Our Proposed Prefetching*	*Simple Stride Prefetching*	*No Prefetching*	*Our Proposed Prefetching*	*Simple Stride Prefetching*	*No Prefetching*	*Our Proposed Prefetching*	*Simple Stride Prefetching*
JPEG Decoder	0.11	0.02	0.02	0.14	0.05	0.09	4.90	0.98	2.04
JPEG Encoder	0.46	0.09	0.15	2.98	0.52	0.97	6.21	0.34	1.23
MPEG Decoder	4.50	1.06	1.89	5.04	1.94	2.75	10.79	4.54	6.98
V44	1.41	0.38	0.54	1.58	0.48	0.56	2.83	0.63	0.84
Sound Recorder	6.61	2.90	3.11	11.90	3.22	5.89	13.09	3.68	7.09

III. EXPERIMENTAL RESULTS

The prefetching mechanism is implemented in five ASIPs. We have chosen these applications as benchmarks since we did not find any publicly available C++ benchmark describing an embedded system. The characteristics of the ASIPs are shown in Table I. The JPEG decoder and encoder are described in [23]. Although both of these programs work on JPEG files, their different access patterns allow us to use them as two different benchmarks. The V44 is a decoder of the modem compression protocol. The MPEG benchmark is an MPEG decoder described in [24]. The Sound Recorder is a sound recorder controller designed according to a case study in [25]. These programs have been synthesized by our synthesis tool [5]. The output of this tool is an object-oriented ASIP in the synthesizable subset of SystemC which contains a processing core, some functional units and a network which connects all these processing elements (see Section 2.1). Since our prefetching mechanism is independent of the implementation type of the methods (hardware or software) we synthesized all class methods of the benchmarks into hardware functional units. The cache is added to the SystemC code of the synthesized ASIPs and some counters in the cache collect the desired information when the ASIPs run the main function of the applications.

The miss ratio of the benchmarks in absence and presence of the proposed prefetching mechanism for various cache sizes are illustrated in Table II.

In order to compare the proposed prefetching mechanism with another mechanism, we also present the results for a simple stride prefetching approach which monitors the data access patterns generated by the methods and starts prefetching upon detecting a stride. In order to detect a stride, this simple strides prefetching approach compares consecutive addresses generated by the processor. Regarding the results presented in Table II, the proposed prefetching method enhances the data cache behavior and causes the miss ratio to reduce on average by 66%. As a result, we can get a higher hit ratio in a cache with a given size or can get a desired hit ratio by a smaller cache. This property is specifically useful in embedded systems due to the limited cache size in such systems.

Table III displays the number of tag comparisons during the execution of each benchmark in a 4-way set associative cache with and without applying the prefetch flag. The cache size and block size are 8K and 8 words, respectively.

If we do not use the prefetch flag, accessing a prefetched data item involves 4 tag comparisons before prefetching and 4 comparisons each time the prefetched data is requested. In the presence of our prefetch flag scheme, however, since accessing a prefetched data before resetting the prefetch flag ideally needs only one tag comparison, the prefetch flag can reduce the number of tag comparisons for prefetched data and alleviate the effect of additional tag comparisons caused by prefetching.

Table III shows that the prefetch flag reduces the tag operations on average, by 21%. This reduction in the tag operations, leads to reduction in prefetching power overhead.

TABLE III. NUMBER OF TAG COMPARISONS WITH AND WITHOUT USING PREFETCH FLAG

Benchmark	# of tag comparisons without prefetch flag	# of tag comparisons using prefetch flag	Tag comparison reduction
JPEG Encoder	33,216,280	25,886,137	22%
JPEG Decoder	24,845,672	21,396,370	13%
MPEG Decoder	27,596,464	18,063,752	34%
V44	79,630,426	61,020,008	23%
Sound Recorder	46,802,764	39,829,511	15%

TABLE IV. THE NUMBER OF REQUESTS THAT HIT/MISS THE CACHE AT THE FIRST PHASE

Benchmark	Cache size=8 kb		
	One-phase accesses	*Two-phase accesses*	*One-phase hit ratio*
JPEG Encoder	5,327,245	197,134	96%
JPEG Decoder	4,430,970	201,987	95%
MPEG Decoder	4,702,425	225,515	96%
V44	14,015,025	204,694	98%
Sound Recorder	8,101,572	101,475	99%

Although this approach reduces the number of tag operations for prefetched data, it increases the access latency, if an address does not hit the cache at the first phase. Table IV presents the number of one phase and two phase accesses in the experiments. The results show that on average, 97% of accesses hit the cache at the first phase. The cache access is done in one phase when either the way determined by the prefetch flag contains the requested data or none of the prefetch flags in a set are set (all ways are accessed in parallel). The fact that we only use the prefetch flag for the prefetched data (which are very likely to be accessed after prefetching) results in a high hit ratio at the first phase. This first-phase hit ratio is larger than the first-phase hit ratios obtained by previous way-prediction mechanisms [16][17]. The details about the area overhead of the proposed prefetch engine can be found in [27].

IV. CONCLUSIONS

In this paper we presented an application specific data prefetching mechanism to improve the performance of object-oriented embedded processors. We also suggested an approach to reducing the power overhead of the prefetching mechanism. Our prefetching approach divides the data accesses of a class method into two conditional and unconditional parts and uses the static information about each class method to prefetch data fields of an object required by a method upon invoking it. This technique improves the miss ratio by 66% on average. Effective management of the access patterns by dividing them based on the method to which they belong and storing the access information of nested loops using a simple structure are the merits of our proposed mechanism. Furthermore, to reduce the prefetch related tag comparisons (and energy consumption) we added a flag to each cache block in order to specify the recently prefetched data item. The cache searches the way whose prefetch flag is set to one and if does not find the data, searches the other ways of a set. The experimental results show that this method reduces tag comparisons by 21% on average. Applying the prefetch flag will result in a two phase cache access and may impose some performance overhead on the system. However, since we use the prefetch flag only for the prefetched data items, which are very likely to be accessed after prefetching, we can get a high first phase hit ratio. Our results show that 97% of the accesses hit the cache at the first phase.

REFERENCES

[1] J.L. Hennessy, and D.A. Patterson, Computer Architecture: a Quantitative Approach, 3rd Edition, Morgan Kaufmann, 2003.

[2] A. Malik, B. Moyer, and D. Cermak, “A Low Power Unified Cache Architecture Providing Power and Performance Flexibility,” Intl. Symp. On Low-Power Electronic Design, June 2000.

[3] S. Segars, “Low Power Design Techniques for Microprocessors,” Intl. Solid-State Circuits Conf. Tutorial, 2001.

[4] Y. Guo et. al. “Energy Aware Data Prefetching for General Purpose Programs,” In Proc. of Power-Aware Computer Systems, Dec. 2004.

[5] M. Goudarzi, S. Hessabi, “The ODYSSEY Tool-Set for System-Level Synthesis of Object-Oriented Models,” Proc. of *Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS V)*, LNCS 3553, Greece, Jul. 2005, pp. 394-403.

[6] M. Modarressi, M. Goudarzi, and S. Hessabi, "Application-Specific Hardware-Driven Prefetching To Improve Data Cache Performance," Tenth Asia-Pacific Computer System Architecture Conf. (ACSAC'05), LNCS 3740, Singapore, 2005, pp. 761 – 774.

[7] J. W. C. Fu, and J.H. Patel, “Stride Directed Prefetching in Scalar Processors,” In Proc. of the 25th Annual Symp. on Microarchitecture, Nov. 1992, pp. 102-110.

[8] S. Kim, and A. Veidenbaum, “Stride-Directed Prefetching for Secondary Caches,” In Proc. of the 1997 Intl. Conf. on Parallel Processing, Aug. 1997, pp. 314-321.

[9] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, “POWER4 System Microarchitecture,” IBM Technical White Paper, 2001.

[10] S.P. VanderWiel, ”Masking Memory Latency with Compiler Assisted Data Prefetch Controller,” Ph.D. thesis, University of Minnesota, 1998.

[11] M. Karlsson, F. Dahlgren, and P. Sternstrom, “A Prefetching Technique for Irregular Accesses to Linked Data Structures,” In *Sixth International Symposium on High Performance Computer Architecture*, France, Jan. 2000, pp. 206–217.

[12] Y. Solihin, J. Lee, and J. Torrellas, “Using a User-Level Memory Thread for Correlation Prefetching,” In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002, pp.171–182.

[13] Z. Zhang and T. Torrellas, “Speeding up Irregular Applications in Shared Memory Multiprocessors: Memory Binding and Group Prefetching,” In Proceedings of the 22nd Intl. Symp. on Computer Architecture, Italy, June 1995, pp. 1–19.

[14] Z. Wang, “Cooperative Hardware-Software Caching for the Next Generation Memory Systems,” PhD thesis, University of Amherst, MA, USA, 2004.

[15] J. Kin, M. Gupta, and W. H. Mngione-Smith, “The Filter Cache: An Energy Efficient Memory Structure,” Proc. of the 30th Annual Intl. Symp. on Microarchitecture, Dec. 1997, pp.184–193.

[16] K. Inoue, T. Ishihara, and K. Murakami, ”Way-Predicting Set-Associative Cache for High Performance and Low Energy

Consumption," Proc. ISLPED'99 Intl. Symp. on Low Power Electronics and Design, 1999, pp.273-275.

[17] N. Bellas, I. Hajj, C. D. Polychronopoulos, and G. Stamoulis, "Architectural and Compiler Techniques for Energy Reduction in High-Performance Microprocessors," IEEE Transactions on VLSI Systems, Vol. 8, No. 3, 2000, pp.317-326.

[18] Yeager, "The MIPS R1000 superscalar microprocessor," IEEE Micro, Vol. 6, No. 2, Apr. 1996, pp 28-40.

[19] M. Goudarzi, S. Hessabi, and A. Mycroft, "No-Overhead Polymorphism in Network-on-Chip Implementation of Object-Oriented Models," Proc. of Design Automation and Test in Europe (DATE'04), February 2004.

[20] M. Goudarzi, S. Hessabi, A. Mycroft, "Object-Oriented Embedded System Development Based on Synthesis and Reuse of OO-ASIPs," Journal of Universal Computer Science, Vol. 10, No. 9, Sep. 2004, pp. 1123-1155.

[21] P. Petrov and A. Orailoglu, "Performance and Power Effectiveness in Embedded Processors: Customizable Partitioned Caches," IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems, Vol. 20, No 11, Nov. 2001.

[22] T.C. Mowry, S. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," In Proc. of the Fifth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, MA, Sept. 1992, pp. 62-73.

[23] M. Najafvand, "Implementation of a JPEG OO-Processor," M.S. thesis, Sharif University of Technology, Tehran, Iran, 2006.

[24] N. MohammadZadeh, S. Hessabi, M. Goudarzi, "Software Implementation of MPEG2 Decoder on an ASIP JPEG Processor," Proc. of Intl. Conf. on Microelectronics (ICM'05), Pakistan, Dec. 2005.

[25] I. Porres, AND O. Lilius, "Digital Sound Recorder: A Case Study on Designing Embedded Systems Using UML Notation," Technical Report No. 234, Turku center, Finland, 1999.

[26] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Multiple access caches: energy implications," *In Proc. of the IEEE CS Annual Workshop on VLSI*, Apr. 2000.

[27] M. Modarressi, "Design and implementation of an object-aware cache for object-oriented ASIPs," M.S. thesis, Sharif University of Technology, Tehran, Iran, 2006.