OBJECT-AWARE CACHE: HIGHER HIT-RATIO IN OBJECT-ORIENTED ASIPs

Maziar Goudarzi, Shaahin Hessabi
Sharif University of Technology, I.R.Iran
gudarzi@mehr.sharif.edu, hessabi@sharif.edu

Alan Mycroft
University of Cambridge, UK
alan.mycroft@cl.cam.ac.uk

Abstract

At any point in time in an object-oriented (OO) program, a class method is running whose set of unconditionally-accessed data fields can be statically determined. We propose to fetch this set prior to or during the method execution to increase the data cache hit-ratio. This requires that either the software directs the processor cache controller, or the processor is aware of the currently running class method. We follow the latter approach by extending our previous work where we introduced the object-oriented application-specific instruction processor (OO-ASIP) as a processor whose instruction-set consists of methods of a class library. Such an OO-ASIP is aware of the currently running method and can therefore prefetch the unconditionally-accessed data fields of the called object to the cache. This is a "directed" hardware prefetching policy compared to "speculative" (pre)fetching of traditional caches. We develop formulas for the hit-ratio and show that this approach results in higher hit-ratio than a traditional cache.

Keywords: *Embedded system; object-oriented ASIP; hardware data prefetch; directed fetching policy.*

1. INTRODUCTION

Embedded systems are attracting more and more attention due to technology advancements as well as market demands. Components efficiency, that has already been important in general-purpose systems, has become even more important in such embedded ones. Cache is one of these components that highly impacts system performance by reducing average data access latency [1]. Any improvement in the cache performance is desirable to improve embedded system performance.

Traditionally, cache controllers retain any accessed data, hoping that it will soon be re-accessed (*temporal locality*). In addition, they *speculatively* fetch the neighbouring data items, hoping that they will also be accessed shortly (*spatial locality*). In a general-purpose system, where nothing can be assumed about the access pattern of programs, this *speculative* policy is inevitable. However, in the special-purpose environment of an embedded system, where the system software is known and remains reasonably unchanged, such assumptions are applicable although still hard to take advantage of. Obtaining static and dynamic access pattern information can be time consuming but is a one-off operation for the system software, and hence, will not be of big significance. This static and/or dynamic analysis can result in a huge amount of information that raises the question of "how to hold this information for the cache controller to take advantage of". In object-oriented software, the decomposition of code into class methods along with the decomposition of data into objects can highly simplify this problem; access-pattern information can be kept at method-level. Class methods may be called on different objects, but the *unconditionally-accessed* data fields of the called object are the same for all invocations. If the cache-controller knows the currently called object and method, or is *object-aware* as we say, it can accordingly prefetch the data items that shall be certainly accessed by the method.

Section 2 reviews our previous work and shows how its cache controller is *object-aware*. We then formulate the problem in Section 3 and analytically prove in Section 4 that prefetching unconditionally-accessed data items improves the hit-ratio. A direct extension of this scheme will also be introduced in Section 4. Finally, Section 5 concludes the paper.

2. OUR PREVIOUS WORK

In previous work [2], we identified method invocation (among objects in an object-oriented model) with instruction execution in a processor. In other words, we view a method invocation on an object as an instruction to be executed specifying the object and additional arguments as instruction operands. This view results in identifying public methods of the class library as the instruction set of a corresponding processor. Such

CCECE 2004- CCGEI 2004, Niagara Falls, May/mai 2004
0-7803-8253-6/04/$17.00 ©2004 IEEE

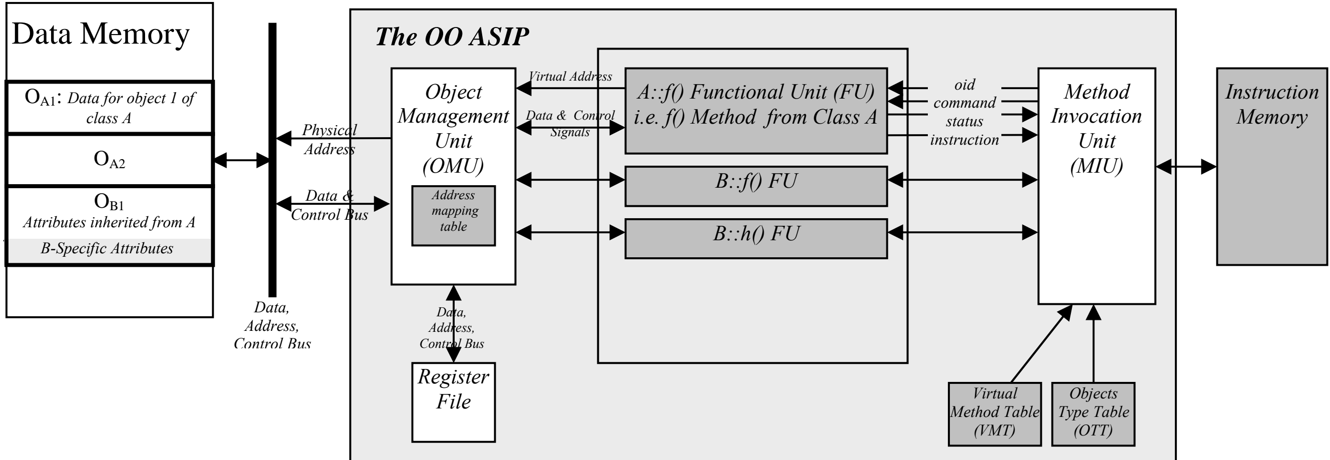

Fig. 1. Internal structure of an OO-ASIP for a class *A* with *f()* method, and class *B* derived from *A* while redefining *f()* and introducing *h()* method. [2]

a processor will be able to execute any program comprised of objects of that class library and even its future extensions; i.e., all applications with that class library. We call this processor an OO-ASIP.

A prototype internal architecture of the OO-ASIP is shown in Fig. 1. The instructions, i.e. method invocation commands, are read from instruction memory by the Method Invocation Unit (MIU). The instruction designates a method (opcode) and at least the called object (operand(s)). Based on the class of the called object (available from the Object-Type Table or OTT), the MIU identifies (through the Virtual Method Table or VMT) and invokes the corresponding Functional Unit (FU) implementing, in hardware, the method that the instruction designated. An activated FU may also invoke other FUs (call additional methods) to accomplish the instruction; this is done by contacting the MIU through “instruction” and “status” signals in Fig. 1.

To access data fields of the object, the FU contacts the Object Management Unit (OMU) which makes the memory transactions. As shown in Fig. 1, objects data are stored in the data memory. Objects of the same class (e.g. O_{A1} and O_{A2} being respectively object 1 and object 2 of class A) have identical layouts for their fields, while objects of derived classes (e.g. O_{B1} being object 1 of class B derived from class A) append the above layout with the newly introduced data fields (the “B-Specific Attributes” part in Fig. 1).

The OMU also contains a data cache and controls it to accelerate data access. The OMU knows which FU is connected to each of its right-hand side ports, and is hence *aware* of the currently running method. The “virtual address” that each FU passes to the OMU consists of an “object identifier” *oid* (already passed to the FU by the MIU) and an “index” that identifies the object data field that the method demands to access; therefore, the OMU is also *aware* of the called object. The OMU, as the cache controller, is *aware* of the called object and method and hence by our definition is *object aware*.

The details on what the OO-ASIP is useful for, how it is synthesized and programmed, and how it implements virtual-method despatch can be found in [2] and [3]. The point that we wish to make use of in this paper is that the OO-ASIP knows enough of the running object-oriented software to be able to implement the directed prefetching scheme that we analyse in this paper.

3. PROBLEM FORMULATION

The additional information, of which we take advantage in this work, is the access pattern of the method implementation to the object data fields; this affects data cache performance and hence all our formulas in this paper apply only to the data cache.

We define the set of all data accesses that a certain method M may use during its execution as A_M. We divide this set into two disjoint subsets according to whether the access is unconditionally done when the method executes, or depends on some run-time condition (e.g. all data accesses in the body of an *if-then-else* statement); the former subset is shown by $A_{M,un}$, and the latter by $A_{M,cnd}$. Obviously, A_M will be the disjoint union of these two subsets:

$$A_M = A_{M,un} \cup A_{M,cnd} \qquad (1)$$

The above factors are static in nature and can be determined from a static analysis of the class method. Moreover, as an OO-ASIP is designed for a given class library, assuming that such information is available to the cache controller shall not violate the validity of the approach. Our analysis involves some other factors that have a dynamic nature; they need a run-time analysis

with some sample inputs to get computed. These are introduced below.

For a sample run of a certain method M, the probability that a certain data access operation "*a*" is performed shall be the ratio of the number of "*a*" accesses to the total number of data accesses performed by M:

$$P_{M,\,a,\,acc} = N_{M,\,a,\,acc} \,/\, N_{M,\,acc} \qquad (2)$$

where $N_{M,\,a,\,acc}$ shows the number of times access to "*a*" was performed during the sample run of method M, and $N_{M,\,acc}$ shows the total number of data accesses that M performed in that same run.

We also denote the probability that a certain data d is available in the cache as $P_{M,\,d,\,avl}$.

The hit ratio of a cache is defined as the ratio of data accesses that are answered by the cache to the total number of accesses. To express this ratio in terms of the individual accesses, we use the above probabilities of performing a certain access and finding it in the cache. This results in the following equation:

$$h = \sum_{\text{for all } M \text{ and } a} P_{M,a,acc} \times P_{M,a,avl} \qquad (3)$$

Finally, we refer to the hit ratio of a traditional cache as h_{tr} and specify the hit ratio of an object-aware cache as h_{oa}. The next section uses these notations to analyse the hit ratio in presence and absence of the *object-awareness* capability.

4. HIT-RATIO IMPROVEMENT ANALYSIS

Hit ratio obviously depends on several factors; e.g. cache organisation, cache size, and replacement policy. To make a concrete analysis, we suppose that all such factors are the same in both cases and the only difference is in the *object-awareness* capability.

Moreover, hit ratio can be determined for the entire execution of a program, or for a certain amount of time, or for a certain piece of code. For the sake of simplicity, we restrict ourselves to the execution of a sample run of a single class method M. Later, we extend the results to an arbitrary time span.

Specialising equation (3) to a single class method M results in the following equation where we have used the M subscript to denote the hit ratio while executing a certain method M:

$$h_M = \sum_{a \in A_M} P_{M,a,acc} \times P_{M,a,avl} \qquad (4)$$

The above equation can be broken into the sum of two summations over *unconditional* and *conditional* accesses as follows:

$$h_M = \sum_{a \in A_{M,un}} P_{M,a,acc} \times P_{M,a,avl} + \sum_{a \in A_{M,cond}} P_{M,a,acc} \times P_{M,a,avl} \qquad (5)$$

Now we get to the comparison between the traditional and object-aware caches. For a traditional cache, no further modification is needed; hence, the hit ratio in a traditional cache when executing a class method M is:

$$h_M^{tr} = \sum_{a \in A_{M,un}} P_{M,a,acc} \times P_{M,a,avl}^{tr} + \sum_{a \in A_{M,cond}} P_{M,a,acc} \times P_{M,a,avl}^{tr} \qquad (6)$$

Note that the access pattern is independent of the cache operation, and therefore, $P_{M,a,acc}$ is the same for both traditional and *object-aware* caches.

For an *object-aware* cache, however, the fetching policy ensures that all unconditional accesses hit the cache since their corresponding data is prefetched upon method invocation; hence, the $P_{M,a,avl}=1$ for all such accesses. This results in the following equation for the *object-aware* cache:

$$h_M^{oa} = \sum_{a \in A_{M,un}} P_{M,a,acc} + \sum_{a \in A_{M,cond}} P_{M,a,acc} \times P_{M,a,avl}^{oa} \qquad (7)$$

Subtracting Equation 6 from 7 gives the improvement in the hit ratio caused by the *object-awareness* capability, assuming that $\forall a \in A_{M,cond}\ P_{M,a,avl}^{tr} = P_{M,a,avl}^{oa}$ (see below):

$$\Delta h_M = \sum_{a \in A_{M,un}} P_{M,a,acc} \times (1 - P_{M,a,avl}^{tr}) \qquad (8)$$

This shows that the *object-awareness* capability shall sure improve the hit ratio unless either M has no unconditionally-accessed data ($P_{M,a,acc}=0$), or the traditional cache organisation is such that unconditional accesses certainly hit the cache ($P_{M,a,avl}^{tr} = 1$). Applications which run exclusively from the cache after an initial start-up transient would satisfy this criterion, but generally cache memory in embedded systems is limited and thus cache-awareness improves behaviour.

Equation 8 shows that the improvement is expectedly independent of the *conditional* accesses; our *object-awareness* scheme, in its simplest form, does not involve and affect such accesses (and hence $\forall a \in A_{M,cond}\ P_{M,a,avl}^{tr} = P_{M,a,avl}^{oa}$) although it is quite capable (and indeed motivated) to be further extended to this subset.

The above analysed scheme only requires a static analysis of the method code to distinguish the data for prefetching. A direct extension, relying on dynamic analysis, would be to prefetch the data that are accessed above a likelihood threshold. Such likelihood information can be obtained from a dynamic analysis of the method using some reasonable test vectors before synthesising the OO-ASIP. If we show this set of accesses as $A_{M,threshold}$, the resulting hit ratio improvement becomes:

$$\Delta h_M = \sum_{a \in A_{M,threshold}} P_{M,a,acc} \times (1 - P_{M,a,avl}^{tr}) \qquad (9)$$

Equation 9 allows a design trade-off by introducing the $A_{M,threshold}$: on one hand, Δh_M can be higher (due to the wider summation) compared to Equation 8; but on the

other hand, it consumes higher memory bandwidth (and more power) to prefetch the associated data, it also requires a bigger cache controller to remember this per-method information, and moreover, some of these prefetched data may not be actually accessed resulting in some wasted resources. This is a trade-off between hit-ratio and area/power/bandwidth overhead controlled by the threshold value: the lower the threshold, the higher the hit-ratio and also the higher the overheads. At one extreme, the threshold can be lowered to zero so that all possibly accessed data are prefetched; this satisfactorily results in $h_M=1$ (since $\forall a \in A_M \; P_{M,a,avl} = 1$ causing Equation 5 to reduce to absolute one) but imposes the highest overhead as well. At the other extreme, the threshold can be increased to 1; this causes $A_{M,threshold}=A_{M,un}$ and reduces the scheme to the previous one in Equation 8.

Up to this point we have talked about the hit-ratio during execution of a single method M. This does not violate the validity of the results for the entire program execution, but brings in a new set of coefficients; the program consists of a set of such method calls and hence the overall hit ratio can be computed by the following equation, where $P_{M,call}$ shows the probability of invoking method M:

$$h = \sum_{\text{for all } M} P_{M,call} \times h_M \qquad (10)$$

Again, since the cache operation is transparent to the program execution, *object-awareness* does not affect $P_{M,call}$. Therefore, the hit-ratio improvement for the entire program execution shall be the following for which the clauses that we stated regarding hit-ratio improvement still hold:

$$\Delta h = \sum_{\text{for all } M} P_{M,call} \times \Delta h_M \xrightarrow{\text{Equation 8}}$$

$$\Delta h = \sum_{\text{for all } M} P_{M,call} \times \Big(\sum_{a \in A_{M,un}} P_{M,a,acc} \times (1 - P^{tr}_{M,a,avl}) \Big)$$

5. CONCLUSIONS

We introduced the *object-awareness* capability as a feature of the cache controller that is "*aware* of the currently called object and method" in the running object-oriented program. Observing that an embedded system is a special-purpose environment where the system software is known and stationary, we proposed to use *object-awareness* to prefetch the data items that shall be certainly referenced by the called method. Our previous work on OO-ASIP enabled us to use its cache controller hardware for prefetching; this avoids the instruction overhead of software prefetching [4] while still offering its full potential. Our analysis proved that object-awareness improves the hit-ratio proportional to the ratio of *unconditional* memory references to the total number of references; hence, the object-aware cache is in some cases much better than a traditional cache with similar characteristics (other than *object-awareness*) and is never less efficient. We also extended the above scheme by proposing to prefetch all those data items that their access likelihood is above a certain threshold. Such prefetches, however, may be wasteful as some of the prefetched items may not be actually referenced. This gives a trade-off between the potential hit-ratio improvement and the power/area/bandwidth overhead.

Various previous works have used software data prefetching [4, 5], hardware data prefetching [6], or a combination of them [7] to direct or hint the cache controller in prefetching data. But to the best of our knowledge, this is the first attempt to customise the cache to a "given" software program.

We are developing a synthesis tool for the OO-ASIP and a compiler for its programs. When these are ready, we shall implement *the object-aware cache* to test it in action and report experimental results.

Acknowledgements

The first two authors wish to thank the Department for Hi-Tech Industries in the Ministry of Industries and Mines for a grant supporting this and related OO-ASIP work.

References

[1] Hennessy, J., and D. Patterson. *Computer Architecture: a Quantitative Approach (3rd Edition).* Morgan Kaufmann, 2002.

[2] M. Goudarzi, S. Hessabi, and A. Mycroft, "Object-oriented ASIP design and synthesis," *Proc. of Forum on specification and Design Languages (FDL'03),* September 2003.

[3] 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.

[4] S.P. VanderWiel, and D.J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys,* vol. 32, issue 2, pp. 174-199, June 2000.

[5] D. Bernstein, C. Doron, and A. Freund, "Compiler techniques for data prefetching on the PowerPC," *Proc. of International Conf. on Parallel Architectures and Compilation Techniques,* June 1995.

[6] K.K. Chan, et al., "Design of the HP PA 7200 CPU," *Hewlett-Packard Journal,* vol. 47, no. 1, February 1996.

[7] S.P. VanderWiel, "Masking memory access latencies with compiler-assisted data prefetch controller," *PhD Thesis,* University of Minnesota, 1998.