Interprocedural Speculative Optimization of Memory Accesses to Global Variables

Lars Gesellensetter and Sabine Glesner

Institute for Software Engineering and Theoretical Computer Science,
Technical University of Berlin, FR 5-6, Franklinstr. 28/29, 10587 Berlin, Germany
`{lgeselle,glesner}@cs.tu-berlin.de`
`http://pes.cs.tu-berlin.de/`

Abstract. The discrepancy between processor and memory speed, also known as memory gap, is steadily increasing. This means that execution speed is more and more dominated by memory accesses. We investigate the use of globals, which reside inherently in memory, in standard applications and present an approach to reduce the number of memory accesses, thereby reducing the effect of the memory gap. Our approach can explicitly deal with uncertain information and, hence, optimize more aggressively with the help of speculative techniques while not changing the semantics of the optimized programs. We present an implementation of the proposed optimization in our compiler framework for the Intel Itanium and show that our techniques lead to an increased performance for the SPEC CPU2006 benchmarks, thus showing that the impact of the memory gap can be effectively mitigated with advanced speculative optimization.

1 Introduction

Over the past decade, program performance has been increasingly influenced by memory system performance rather than CPU speed. This phenomenon, termed *memory wall* or *memory gap*, was foreseen in the 1990s [WM95] and is due to the fact that, as technology evolves, CPU speed is increasing faster than memory speed. This can have severe consequences, e.g. modern processors like the Intel Itanium stall up to 50% of the time during program execution. While this effect may not be that drastic for all classes of applications, e.g. for scientific computations on arrays, it poses a challenge for general-purpose applications. This is especially severe if complex data structures on the heap are used, because then the cache cannot mitigate all the effects of the memory system. Novel optimization techniques are therefore required to overcome the memory wall.

In this paper, we consider the class of memory accesses induced by the use of global variables. The concept of globals is widely used across programming languages. In languages with an explicit notion of memory, like C, globals generally lead inherently to memory accesses. Since all functions of a program may read or even change a global, they cannot be safely kept in a register across a call. Our goal is to optimize this class of memory accesses in order to reduce the induced memory traffic and to increase overall program performance. The effect of an optimization for a given program can be measured by the number of issued loads and stores during program run-time on the one hand, and by the overall run-time performance on the other.

Our solution is based on the idea that the memory gap can be reduced if we preload data from memory early enough. While in the classic approach, this idea is constrained by the numerous dependencies between memory accesses, speculative techniques allow for more optimization potential: By speculating about which memory dependencies will be actually present at run-time, irrelevant false dependencies can be neglected. This leads in general to more optimization and to a substantial increase in performance. Certain processors, e.g. the Intel Itanium, offer hardware support for speculatively executing instructions. We have exploited this feature in the work presented here.

In this paper, we present an algorithm to reduce the memory overhead induced by the use of global variables. Our algorithm works in two stages. First, we perform a global program analysis on global usage and use this information to keep globals in registers as long as possible. Secondly, we extend this algorithm by enriching the analysis with results from static branch prediction, and by optimizing more aggressively with the help of speculation, while still strictly preserving program semantics. As a case study, we implemented the optimization in our compiler framework and performed measurements on the Intel Itanium platform using the SPEC 2006 benchmark suite. We show that run-time improves significantly in many cases, and even more with the speculative variant.

The rest of this paper is organized as follows: In Section 2, we present empirical results on the actual use of globals in the SPEC benchmarks and introduce the concept of speculative optimizations. Section 3 presents our global program analysis for the use of global variables. The optimizations based on this information are presented in Section 4. In Section 5, we describe our case study, where we implemented the optimization for the Intel Itanium, and present the results. Related work is reviewed in Section 6, and we end with a conclusion in Section 7.

2 Background

2.1 Globals in the SPEC2006 Suite

We analyzed the use of global variables in the SPEC2006 benchmark suite to investigate the relevance of globals in standard application programs. The number of globals defined in the source programs ranges from a few to over thousand. A significant fraction (up to 75%) thereof is constituted by globals of integer and pointer type. The number of used globals correlates mainly with program size, and is similar for integer and floating-point programs, respectively.

We performed typical program runs using the *train* data of the SPEC benchmarks to measure the dynamic usage of globals at run-time (see Tab. 1). Using the instrumentation tool Pin [LCM$^+$05], we counted all read and write accesses to main memory, resp., and compared them to the total number of issued instructions. We then distinguished between accesses to heap, stack, and global data. The latter were further broken down by different data types. We see that the amount of memory accesses ranges from 10% to almost 40% of all instructions. More specifically, in some cases accesses to globals constitute a significant fraction of all memory accesses (up to 50% for sjeng, over 20% for perlbench, still significant for gcc, gobmk, h264ref, and milc). Especially, integer globals are often the major contributors (sphinx3, milc, sjeng, perlbench, bzip2, gcc).

2.2 Speculation on Data Dependencies

On modern processors, a load can cause the processor to stall up to 200 cycles. To mitigate this problem, loads can be moved upwards to hide their latency, or redundant loads can be eliminated. In doing so, one has to consider the dependencies among the instructions. Speculative techniques can be used to neglect memory dependencies and thus provide more potential for optimization. An example is given in Fig. 1. On the left (Fig. 1(i)), a value is loaded into register $r9$ and then used. The load can entail a long stall. Moving up the load is not safe in general since $r8$ and $r11$ may refer to the same address. However, if we have some evidence that the addresses are different, we can optimize speculatively (see Fig. 1(ii)). We move the load across the store and make it an *advanced* load. Then, after the store, we check whether the loaded value is still valid. If so, we avoided a long stall. Otherwise, we simply reload it again, and the program runtime is similar to the left program. It is important to note that speculation does not sacrifice correctness. If we misspeculate, we only get additional overhead.

In the example, we assumed hardware support for speculation, which allows for efficient checks whether the speculation was correct. On the Itanium, this is done by the *ALAT (Advanced Load Address Table)*, which collects the addresses of advanced loads. Whenever a store conflicts with one of the entries, it is removed from the table. Thus a validity check simply checks whether the corresponding address is still in the ALAT. Alternatively, this can be carried out solely by software. Then, for all intervening stores, additional instructions must check whether the addresses overlap, and recovery code has to be added to reload the

Table 1. Number of executed instructions, fraction of accesses *(reads/writes)* to memory at run-time, split up by accesses to heap, stack and globals, the latter split up again by the different datatypes ('0' close to 0, '–' really 0)

		ins	% ins	% of memory			% of global				
	Benchmark	(10^9)	mem	heap	stack	global	int	ptr	float	arr	other
INT	perlbench	4.86	8.6/2.1	78/66	1.7/5.1	21/29	26/25	68/68	–/–	2.1/1	4.3/5.3
	bzip2	47.3	14/4.4	85/58	14/42	0.7/0	26/0.1	–/–	–/–	74/100	–/–
	gcc	6.36	13/4.1	82/59	4/9.2	14/33	19/1.6	14/0.2	–/–	50/69	17/30
	mcf	4.77	25/3	98/99	1.6/1.1	0.5/0.1	–/–	–/–	–/–	–/–	100/100
	gobmk	101	15/4.5	76/72	17/28	6.5/0.5	0/0.3	0/0	–/–	94/73	6.4/27
	hmmer	29.2	18/4.2	91/100	8.3/0	0.3/0	0.6/2.6	–/–	–/–	99/97	–/–
	sjeng	26.6	15/3.4	30/9	18/40	52/51	34/38	–/–	0/0	66/62	0/0
	h264ref	142	21/2.3	70/51	24/39	5.9/10	0.2/0.2	35/0	–/–	0.9/0.9	64/99
FP	milc	35.3	27/11	77/52	19/48	3.9/0.01	50/0.1	27/0	0/95	23/4.9	0/0
	lbm	11,0	8.4/5.1	100/100	0/0	0/0	–/–	–/–	–/–	–/–	–/–
	sphinx3	8,96	13/0.9	98/90	1.8/7.5	0.2/2.8	100/100	0/0	–/–	0/0	–/–

```
                         ld.adv r9 = [r11]           ld.adv r9 = [r11]
      ...                     ...                         ...
st [r8] = r12            st [r8]     = r12           st [r8]     = r12
ld r9   = [r11]          ld.chk r9 = [r11]           cmp          r8,r11
   STALL                                             beq recovery_code
...     = r9             ...         = r9      back: ...          = r9

      (i)                      (ii)                       (iii)
```

Fig. 1. Speculative optimization of memory accesses

value if necessary (see Fig. 1(iii)). This leads to a larger overhead, but can still pay off if the expected gain is sufficiently high. The proposed approach of this paper can be used for both alternatives. It is only reflected in the cost model which way is chosen. We performed a case study on the Intel Itanium, which has hardware support for speculation.

3 Analysis of the Usage of Globals

In this section, we describe the analysis of the usage of globals. The optimization needs to know which pieces of code might use or change the value of a global. While this is directly known for simple instructions, it is not for function calls. We first describe a basic analysis collecting the information which globals are affected by a function. Then we make the analysis more precise and also consider information from static branch prediction. This information will later be used to decide about applicability and profitability in the speculative optimization. Finally, we briefly present the characteristics of the SPEC2006 benchmarks.

3.1 Basic Analysis on Globals

The basic analysis is a straight-forward interprocedural data flow analysis (see e.g. [NNH99]). We first determine for every function, which globals are directly used and modified. Then we propagate this information along the call graph and calculate the fixed point. At the moment, only unaliased globals are considered, hence it is evident whether or not a statement changes a global. If a function call cannot be resolved statically, we assume that all globals are possibly changed by that call. We finally get for every function the list of affected globals.

3.2 Extended Analysis

The analysis presented so far is a quite conservative estimation of the global usage. In this section, we make it more precise. First, we determine which globals definitely must be changed by a function. These are globals that are modified in post-dominators of the start block of a function. This information is propagated across function calls that are definitely (i.e., unconditionally) executed (yielding *MustDef(f)*). Next, we consider potential changes of globals. We have implemented a state of the art static branch predictor, following [WL94]. This yields for every basic block a static estimate of its execution frequency (relative to function invocation). We use this to attach a frequency to every access to a global. Initially, for every function f and every global g, we collect the maximum frequency of g's use and definition within f (*UseFreq(f,g)*, *DefFreq(f,g)*). We also collect the maximum frequency for a function call from f to f' (*CallFreq(f,f')*). In a fixed point iteration over the call graph, we propagate the frequencies: The frequency *UseFreq(f,g)* is updated at a call to a function f' if the resulting frequency *CallFreq(f,f')* $\times$ *UseFreq(f',g)* is higher than the previous value.

3.3 Analysis Results

The analysis gives us for every function a list of globals that may be affected by a call to it. For the extended analysis, this information is annotated with the estimated frequency, and we also know which globals are definitely changed. This raises the opportunity for speculation: If a global is *not* definitely changed by a function call, we can speculate that it remains unchanged. The frequency can be used for cost estimation. For the SPEC benchmarks, on average 23 globals may be changed by a function. However, only 1.2 globals are definitely changed. The discrepancy between these numbers constitutes the opportunity for speculation.

4 Optimization

The optimization keeps selected globals in registers throughout a function to reduce memory accesses and to avoid stalls. With the results from the previous section, we know to which extent globals are affected by function calls. This information is required to decide where to insert compensation code (synchronizing the global's value with memory), which influences the profitability.

4.1 Overview

The optimization considers each function in turn, identifies the globals used within the function and estimates the performance gain. Then it selects the best candidates and performs the actual transformation. We present two versions of the optimization: In the basic version, no speculation is used. This optimization can be used on any architecture that can spare registers for optimization. The extended version uses speculation to reload globals only if necessary.

Identify Candidates. Candidates are all globals that are used within the function and have integral data type (i.e. fit in a register).

Rate Candidates. For all candidates in turn, the optimization is performed virtually. We collect the information where compensation code (i.e. loads and stores) has to be placed for a given global (see Sec. 4.2). We then have the following parameters to calculate the estimated gain of the optimization: The number of uses and definitions of the global (*UseCount/DefCount*), which represents the number of loads and stores that could be removed by the optimization, and the number of newly introduced loads and stores (*LoadCount, StoreCount*), which keep the global valid at function call borders. All counts are weighted by the estimated frequency of the corresponding block. The score is then calculated by the following weighted sum (with coefficients $w \in \mathbb{R}$):

$$score = w_{use} \cdot UseCount + w_{def} \cdot DefCount + w_{ld} \cdot LoadCount + w_{st} \cdot StoreCount$$

Since *UseCount* and *DefCount* correspond to the gain, the corresponding coefficients will be positive. On the other hand, *LoadCount* and *StoreCount* indicate the introduced overhead, hence their coefficients will be negative. In case of the speculative optimization, the newly introduced loads will be further broken down into *advanced* and *check* loads (with corresponding weights w_{lda}, w_{ldc}). The score function together with the weights constitutes the cost model.

Select Best Candidates. Only candidates that have a score exceeding a given threshold are considered. Since every selected candidate will eventually require a fixed register, only the best *MaxGlob* candidates are chosen per function.

Perform Optimization. This step actually performs the transformation by replacing all occurences of the global by references to the selected fixed register and by inserting compensation code, as determined before.

4.2 Placement of Compensation Code

The optimization keeps a global in a register throughout the complete function. To ensure correctness, it has to make sure that on the one hand the correct value is in the register whenever it is used internally in the function, and on the other hand that the correct value of the global is in the memory whenever a function is called that may use it. Those two tasks are actually dual to each other, as we see in the following. The optimization uses the results from the analysis presented in the previous section to determine the effects of a statement w.r.t. a global.

Effects of Statements. A statement can have the following effects w.r.t. a given global: The global may be used (*USE*) or defined (*DEF*) directly by the statement. Furthermore, the statement may make a call to a function in which again the global could be used (*XUSE*) or defined (*XDEF*), this time externally.

Compensation Code for Internal Uses. First, for all uses of the considered global in turn, a backwards traversal of the CFG is started. It considers all possible paths leading to the use and stops only on statements that define the global's state, i.e. a *DEF/USE* (in register) or *XDEF* (in memory). The search yields a tree of basic blocks. Fig. 2 gives an example. Starting from a use of *glob*, a tree consisting of all possible paths to this use is constructed, stopping at (internal or external) definitions of *glob*. Functions *f* and *g* can possibly change *glob*. Nodes

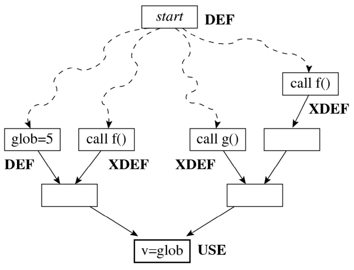

Fig. 2. Constructed Tree

that have an effect on *glob* are annotated correspondingly. Note that the first basic block of a function has always a *DEF* mark since initially the global is loaded from memory. To decide whether the global is available at the use site, we propagate these marks from leaves to root. If all children of a node have the same mark, it is also set for the parent. Otherwise, in all children with *XDEF*, a load instruction is inserted, and they receive now the *DEF* mark, as does the parent. Thus the global is now in a register. If the *XDEF* mark is propagated to the root, a load instruction is inserted direct before the regarded use, and the root node is marked accordingly.

Fig. 3 illustrates the marking algorithm for our example. Left, we see the initially constructed tree with its marks, starting from a use of the global at the root. In the middle, we see an intermediate step. The left child has two children with different marks, hence it is marked with *DEF*, as well as its right child, in which also compensation code is inserted. The right child has only children with *XDEF* marks and hence gets the same mark. The rightmost figure shows the final step. Both children of the root have different marks, hence the root as well as its right child get the *DEF* mark, and compensation code is again inserted in the right child. From now on, the modified nodes will keep their *DEF* marks.

Compensation Code for External Uses. To cope with the dual case, namely to ensure that the correct value of a global is in memory when used by a called function, we proceed almost equally. Now we consider all external uses of the global in turn. We construct a tree of basic blocks as before, and whenever we encounter siblings with different marks, we convert *DEF* nodes to *XDEF* nodes by inserting a store and change the mark of the corresponding parent to *XDEF*. Hence in the end, we receive a tree where the root is marked with *XDEF*.

Aliased Globals. Our approach can be easily extended to aliased globals by treating aliased accesses to the global as *XUSE*/*XDEF*.

4.3 Speculative Compensation Code

As we have seen before, function calls may affect many globals, but only in very few cases this happens definitely. Hence in many cases, the reload of a global after a function call is not necessary and can be made speculative. On the other hand, the cost of misspeculation has to be considered thoroughly. If p is the probability that a given global was changed by a function call, *lat* the load latency, and *mis*

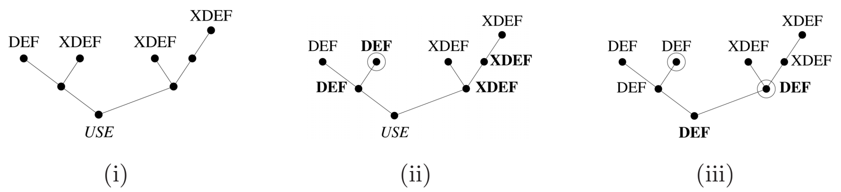

Fig. 3. Example for the different steps of the marking algorithm. Changes are shown boldface, insertion of compensation code is indicated by circles.

the misspeculation penalty, the expected gain will be $gain = (1-p) \cdot lat - p \cdot mis$. We use the previously collected frequencies to approximate p.

The tree is constructed as before, except that we now distinguish between *XDEF* and *XMUSTDEF*. The latter is used to mark external definitions that are definitive or have a frequency above a given threshold *MaxSpecFreq*. *XDEF*, on the other hand, indicates now that speculation should be used. Additionally, the estimated block frequencies are annotated. When a node has only children with *XDEF* and *XMUSTDEF* nodes, it receives the mark for which the sum of the corresponding children's frequencies is maximal. When a node has children both with *DEF* and *XDEF/XMUSTDEF* marks, loads are inserted appropriately as before, but speculation is used only in nodes marked with *XDEF*. All formerly regular loads become *advanced* loads since they initiate the bookkeeping required to check for misspeculation. For speculation, *check* loads are used.

In summary, the results from our analysis can be used for reducing the number of memory accesses induced by globals. We have presented a cost-aware algorithm for the optimization. The cost model allows the algorithm to be applied to different settings, e.g. hardware vs. software supported speculation. Results from a case study are described in the next section.

5 Case Study: Speculation on the Intel Itanium

5.1 Implementation

We implemented the optimization in our compiler framework, which is based on the compiler development system CoSy® by [ACE]. CoSy includes an extensive set of optimizations and the backend generator BEG [ESL89], for which we developed a specification for the Itanium architecture. The Itanium supports speculation by special hardware. A successful *check* load takes one cycle, opposed to 1/5/12 minimum latency for L1/L2/L3 cache accesses (cache size: 16k/256k/1.5M). Misspeculation costs a penalty of 10 cycles extra, plus the time needed to reload the value. Thus speculation has to be used carefully.

5.2 Results

We measured all SPEC CPU2006 C benchmarks[1]. For all measurements, we took the median of three runs. All improvement is compared against the `-O4`-setting of the CoSy compiler, which includes over 50 standard compiler optimizations.

Fig. 4 shows the results using the reference data of the SPEC CPU2006 benchmarks. At first glance we see that the base optimization leads to an improvement in many cases, and that the speculative variant significantly adds further improvement. For gobmk and hmmer, no speculation is done and the optimization merely leads to prefetching of globals, hence both variants perform equally. In the other cases, the misspeculation rate is low and shows that our cost model worked out well. Concludingly we see that speculative reloading outperforms blind re-loading (as in the base optimization).

[1] Except for 462.libquantum since the CoSy frontend is not fully C99 compliant.

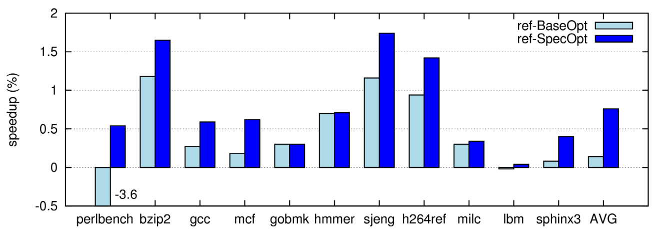

Fig. 4. Results of the SPEC CPU2006 benchmarks

6 Related Work

The presented approach performs register promotion for globals. Much work has been done on register promotion in general. [Wal86] and [CH90] present approaches to global register allocation, which perform register promotion. [CL97] propose register promotion as a separate optimization and focus especially loops. [LCK$^+$98] and [SJ98] exploit partial redundancy to perform register promotion. [BGS99] also focus on eliminating partial redundancy. They investigate how good candidates for register promotion can be statically predicted. The approaches presented so far require conservative, safe information about dependencies, which means especially for memory accesses a very rough overapproximation. This can be solved with speculative techniques. [PGM00] propose a hardware extension that allows to speculate on dependencies and to issue run-time checks to ensure correctness. [LCHY03] present the speculative extension of [CL97]. They report performance improvements for the Intel Itanium. The work discussed so far tries to optimize memory accesses in general. Our presented work, in contrast, tackles a certain class of memory accesses, namely those induced by globals, and therefore can make use of a more precise analysis.

There are also approaches which propose register promotion for global variables. [S090] consider inter-procedural register allocation and also optimize globals. Results from simulation report a significant improvement, but without regarding cache behavior. [CC02] also examine the effects of promoting globals to registers, mainly focusing on reducing power consumption. They perform an architectural exploration to investigate how many separate registers are required for the optimization, and find that already 4–8 registers are sufficient. Differing from our approach, both works promote a global to a register for the whole program, and they do not consider speculative techniques.

7 Conclusion

As it has been foreseen in the past, the memory gap has become a major limiting factor on general-purpose architectures. Hence novel optimization techniques have to be developed to mitigate this effect. Speculation on data dependencies is required to allow more aggressive optimization, and modern architectures are offering efficient support for it. With the work presented in this paper, we have done a step towards that direction. We tackled a certain class of memory accesses, namely those induced by global variables. In our experiments, we could show that our optimization leads to significant improvement on some benchmarks, and this effect is even stronger for the speculative variant.

For the future, we plan to go one step further and consider a broader class of memory accesses. We think that on the one hand, more precise analyses are required that try to make use of the rich information provided in the intermediate representation and also can yield unsure, or speculative, information, and on the other hand, novel optimizations have to be developed that make use of this information and have a precise, architecture-dependent cost model to achieve an overall performance improvement, while preserving the semantics of the program. We think that this will be the key to sustainably reduce the impact of the memory gap on general-purpose applications.

References

[ACE] Associated Compiler Experts bv., Amsterdam, The Netherlands, `http://www.ace.nl`

[BGS99] Bodik, R., Gupta, R., Soffa, M.L.: Load-reuse analysis: design and evaluation. In: PLDI (1999)

[CC02] Cilio, A.G.M., Corporaal, H.: Global variable promotion: Using registers to reduce cache power dissipation. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304. Springer, Heidelberg (2002)

[CH90] Chow, F.C., Hennessy, J.L.: The priority-based coloring approach to register allocation. ACM Trans. Program. Lang. Syst. 12(4) (1990)

[CL97] Cooper, K.D., Lu, J.: Register promotion in C programs. In: PLDI (1997)

[ESL89] Emmelmann, H., Schröer, F.-W., Landwehr, L.: Beg: a generation for efficient back ends. In: PLDI (1989)

[LCHY03] Lin, J., Chen, T., Hsu, W.-C., Yew, P.-C.: Speculative register promotion using advanced load address table (ALAT). In: CGO 2003. IEEE, Los Alamitos (2003)

[LCK$^+$98] Lo, R., Chow, F., Kennedy, R., Liu, S.-M., Tu, P.: Register promotion by sparse partial redundancy elimination of loads and stores. ACM SIGPLAN Notices 33(5) (1998)

[LCM$^+$05] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI (2005)

[NNH99] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)

[PGM00] Postiff, M., Greene, D., Mudge, T.: The store-load address table and speculative register promotion. In: Proceedings of ACM/IEEE MICRO (2000)

[SJ98] Sastry, A.V.S., Ju, R.D.C.: A new algorithm for scalar register promotion based on SSA form. In: PLDI (1998)

[S090] Santhanam, V., Odnert, D.: Register allocation across procedure and module boundaries. In: PLDI (1990)

[Wal86] Wall, D.W.: Global register allocation at link time. SIGPLAN Not. 21(7) (1986)

[WL94] Wu, Y., Larus, J.R.: Static branch frequency and program profile analysis. In: Proceedings of ACM/IEEE MICRO (1994)

[WM95] Wulf, W.A., McKee, S.A.: Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News 23(1) (1995)