Fast and Accurate Evaluation of Memory Performance Upper-Bound

Grigori Fursin, Mike O'Boyle
University of Edinburgh, UK

Olivier Temam, Gregory Watts
Paris South University, France

Abstract

In many cases, compiler-based optimizations for memory performance tuning are too limited and programmers have to resort to manual optimization techniques. However, this process is tedious and time-consuming, especially since there is no indication on when the process can stop, i.e., when optimal memory performance has been achieved or sufficiently approached. Architecture simulators can provide such information but designing an accurate model of an existing architecture is very difficult, and simulation times are excessively long. In this article, we propose a technique that is both fast and reasonably accurate for estimating the memory performance upper-bound of many scientific applications. This technique has been tested on several programs and can be used to guide a manual optimization process, or even to drive iterative compilation techniques.

1 Introduction

Memory optimization can bring significant performance improvements but compiler memory optimizations are still very limited because of the *increasing complexity of memory and processor architecture*. Up to now, compilers assume a rather simplified version of the memory hierarchy: registers and the first-level cache. Some other components are naturally considered in production compilers but the model implicitly used is necessarily far less complex than a full-blown memory system which includes: registers, different cache levels, TLB, write-buffers, buses, main memory, interactions with the operating system and between different processes. Moreover, interprocedural locality analysis is still in its infancy, so the scope of optimization techniques is often restricted to a single procedure. Overall, memory architecture conscious optimization of a program remains limited.

Therefore, whenever performance is at stake, *manual program optimization* remains a necessary complement for improving performance. Economics can also require that a program be finely optimized, e.g., a 25% execution time reduction on a critical application run on a farm of 100 workstations (or an SMP with 100 processors) would require 25 less workstations or processors; with metacomputing, a 25% gain on an application can translate into hundreds or thousands less machines necessary. In practice, manual optimizations, as performed by hundreds of engineers, is a trial and error process: a program transformation is applied, the program performance is evaluated, a new transformation is applied after analysis and so on. This process can be long and tedious, requiring days, weeks or months depending on the program and the performance goal. While this is acceptable in an academic environment, "engineer months" are very expensive in industry. Consequently, companies would basically invest into a task without clear indication on how long it will take, when it should stop and what is the potential return on investment.

These very practical considerations actually translate into a technical issue: we must find a way to estimate beforehand the potential benefit of memory program optimization, i.e., how much the execution time will improve after the whole transformation process is performed. While it is difficult to provide an accurate value of the expected execution time beforehand, we can seek a lower bound of the execution time. Moreover, if we can obtain this lower bound quickly, we can compute it at each step during the iterative manual optimization process, and decide whether we are close to the execution time lower bound or if it is worth carrying on.

A program memory performance upper-bound can be defined here as a program with no cache misses. Other memory parameters than misses naturally play an important role in program performance, but most optimizations target misses [19, 7, 16], so an upper bound must focus on this criterion if it is to be used to drive an iterative optimization process. Note also that the minimum number of misses in a program is not zero (i.e., "no miss") but the number of compulsory misses; however, the fraction of compulsory misses is almost always negligible compared to capacity and conflict misses [10, 15]. Until recently, deducing the *no-miss* execution time from the normal execution time would have been rather easy using hardware counters [2]: the execution time minus the number of misses (as recorded by hardware counters) times the latency would provide with the optimal execution time. However, superscalar processors now have non-blocking caches, out-of-order execution and complex memory hierarchies [11] which all make it impossible to deduce the *no-miss* execution time based on the normal execution time and the number of misses.

Processor simulators, like SimpleScalar [5], provide a simple means to compute this memory performance upper-bound: it is trivial to modify a processor simulator so that it mimics a perfect cache behaviour. However, processor simulators have two severe drawbacks:

(1) they only model the *processor* while the whole system can have a strong impact on memory performance: the way the TLB is reloaded, the bus arbitration mechanism, the physical to virtual mapping in lower cache levels, the type of memory (SDRAM, DDRAM), cache interferences between several processes run concurrently and numerous other specific architecture-dependent issues. Consequently, we need a *system* simulator rather than a *processor* simulator. First, system simulators like SimOS [17] are far less widespread and mature than processor simulators. Second, it is already very difficult to develop a processor simulator that accurately models an existing processor without privileged access to the processor internal workings [9], so that an accurate system simulator would require a huge effort to accurately model the chipset, the memory chips, the operating system and all other components.

(2) a processor simulator is extremely slow: a simulated program on a current superscalar processor is several hundreds times slower than the normal execution [5]. On a system simulator a 2000-fold slowdown or more is likely. Whether the simulator is used only once at the beginning of the optimization process or worse, at each step, such a slowdown is barely acceptable for many programs and not tolerable for applications whose execution time exceeds a few minutes (which is the case for many performance-critical applications).

Consequently, because we need to take into account the whole system architecture, and because we cannot afford excessively long delays, simulators do not provide a satisfactory means, for our purposes, for computing the performance upper-bound. In this article, we propose a technique that is both fast and reasonably accurate for estimating the memory performance upper-bound of a class of programs, where control structures are do-loops and data structures are arrays, which is the case for many performance-critical scientific applications. This technique has been tested on several programs and our final goal is to implement into a production compiler so that it can be used for manual optimization purposes. This research was part of the European Esprit projet MHAOTEU (Memory Hierarchy Analysis and Optimization Tools for the End-User).

2 Principles

The general idea of our technique is to modify the program so that it retains almost all the characteristics of the original program but induces almost no misses. Therefore, the execution time of the instrumented program would provide the performance upper-bound of the original program once all cache misses have been solved.

Removing misses. In a program where do-loops and arrays dominate, almost all misses are due to array references within loops. Let us assume now that all these array references are transformed into scalar references. Then the memory footprint of the resulting program would be negligible compared to the original footprint and the cache size, and the number of misses would be close to 0. The baseline of our technique is to transform each individual array reference into a scalar reference. The real challenge is to make sure that this transformation will not affect the rest of the program characteristics and its execution on a superscalar processor.

Let us consider array reference `A[i]` in the following loop:

```
DO i = 1, N
  ... = A[i]
ENDDO
```

After compiling on a Compaq Alpha EV6, this reference would be translated into the following assembly code:

```
........
lda
$19, 8($19)
ldt
$f13, ($19)
........
```

where register `19` contains the current target address of the load instruction, i.e., the base address of array `A` plus loop counter `i` times the size of one memory element (8 bytes in this example); `lda` is a misleading acronym, it is not a load instruction but an add instruction dedicated to address computations. So in this case, it increments register `19` by 8 to fetch the next element of array `A`. The load instruction is `ldt` which will fetch the data located at the address stored in register `19` into register `f13`. These two instructions combined correspond to array reference `A[i]`.

Assume now that we modify the `lda` instruction as follows:

```
........
lda
$19, 0($19)
ldt
$f13, ($19)
........
```

where only the offset has been modified: the 8 has been replaced with a 0. All the instructions are the same, the same number of computations is performed and the register dependences have not been modified. But now the address referenced by instruction `ldt` *is constant* over the whole loop execution. Consequently, the memory footprint of reference `A[i]` is reduced from $N \times 8$ bytes to just 8 bytes. Considering the minimum cache size is around 8 kbytes, and that the number of references is usually much smaller than 1000 in do-loops, the memory footprint of do-loops where array references have been transformed as above will almost always fit in cache and then only induce as many cold-start misses as the number of array references in a loop, which is negligible.

Correct code execution. Naturally, once the code has been transformed as above, it does not execute properly anymore. Therefore, we make a copy of each procedure: for each procedure `proc`, we create a procedure `procmpc` which is then instrumented as explained above. First, the instrumented procedure is executed, then the original procedure is normally executed to enable normal program execution. However, the instrumented procedure can still modify variables used by the original procedure, so we add a *backup* and a *restore* procedure respectively before and after the instrumented procedure. The purpose of these procedures is simply to backup and then restore all variables (constants and array references) that will be modified by the instrumented procedure. Recall that array references become constants, so that only a small fraction of the original data set is affected and needs to be backuped/restored. Finally, the execution time of the program without misses, i.e., *the upper-bound*, is obtained by summing the execution time of all instrumented procedures and ignoring the normal, backup and restore procedures.

Consider procedure `calc2` from the SpecFP2000 program SWIM. The original code is the following:

```
      SUBROUTINE CALC2
...
        DO 200 J=1,N
          DO 200 I=1,M
            UNEW(I+1,J) = UOLD(I+1,J)+
            UNEW(I+1,J) = UOLD(I+1,J)+
     1      TDTS8*(Z(I+1,J+1)+Z(I+1,J))*(CV(I+1,J+1)+CV(I,J+1)+CV(I,J)
     2      +CV(I+1,J))-TDTSDX*(H(I+1,J)-H(I,J))
            VNEW(I,J+1) = VOLD(I,J+1)-TDTS8*(Z(I+1,J+1)+Z(I,J+1))
     1      *(CU(I+1,J+1)+CU(I,J+1)+CU(I,J)+CU(I+1,J))
     2      -TDTSDY*(H(I,J+1)-H(I,J))
            PNEW(I,J) = POLD(I,J)-TDTSDX*(CU(I+1,J)-CU(I,J))
     1      -TDTSDY*(CV(I,J+1)-CV(I,J))
  200 CONTINUE
...
```

The transformed code is the following, where `calc2mpc` is the instrumented procedure:

```
      SUBROUTINE CALC2
...
        CALL CALC2SAVE
        CALL CALC2MPC
        CALL CALC2RESTORE
...
        DO 200 J=1,N
          DO 200 I=1,M
            UNEW(I+1,J) = UOLD(I+1,J)+...
...
  200 CONTINUE
```

`calc2mpc` is instrumented at the subroutine level, and only the following assembly instructions have been affected in the inner loop body, in addition to similar instructions in the outer loop. These instructions correspond to the above mentioned array references.

```
...
.loc 1 317
lda       $8, 0($8)
lda       $6, 0($6)
lda       $17, 0($17)
lda       $18, 0($18)
lda       $19, 0($19)
lda       $20, 0($20)
lda       $21, 0($21)
lda       $5, 0($5)
.loc 1 321
lda       $16, 0($16)
.loc 1 322
...
.loc 1 317
lda       $7, 0($7)
...
```

The backup and restore procedures are directly written in assembler because a single source array reference can result in multiple load/store instructions. The procedures are shown below:

```
#mpc_start_save

ldiq $2, 0x141b32368
ldq_u $3, ($2)
ldiq $2, 0x14be5f1c8
stq_u $3, ($2)

...

ldiq $2, 0x145196d80
ldq_u $3, ($2)
ldiq $2, 0x14be5f580
stq_u $3, ($2)

#mpc_end_save
```

calc2save

```
#mpc_start_restore

ldiq $2, 0x14be5f1c8
ldq_u $3, ($2)
ldiq $2, 0x141b32368
stq_u $3, ($2)

ldiq $2, 0x14be5f580
ldq_u $3, ($2)
ldiq $2, 0x145196d80
stq_u $3, ($2)

#mpc_end_restore
```

calc2restore

3 Experiments

3.1 Experimental Framework

All experiments were performed on a Compaq Alpha EV67. The Alpha EV67 is clocked at 667MHz, and it is 4-way superscalar, out-of-order, with an 8-Mbyte second-level cache.
For our experiments, we have instrumented the most time-consuming routines of four SpecFP2000 programs: `171.swim, 172.mgrid, 173.applu` and `183.equake`; all programs were run using the Spec `ref` data set. On a first pass, we copied each routine, inserted the calls to the backup, instrumented and restore routines, and the assembly-level modifications were performed on a second pass. We have developed an automatic tool for both passes. However, all instrumentations could be easily performed in a production compiler where all the necessary information is already well-known (location of array references, and expressions of the subscripts), so adding the upper-bound computation capacity in a production compiler would be rather simple. Note also that the way the instrumentation is performed, a single run is sufficient to get both the normal program results and execution time as well as the upper-bound execution time, with a reasonable overhead (at most a slowdown of 2).

3.2 Experimental results

We show in the table below the upper-bound IPC and the expected IPC for all the instrumented procedures. We can see that the expected percentage of improvement varies wildly, and that in some cases, it is very high. For instance, the potential speedup for SWIM is 3.19 while for MGRID it is only 1.39. Obviously, the memory system has a tremendous impact on SWIM performance. Hardware counters [2, 8] would provide this information just as fast but they cannot *quantify* the performance gain that can be achieved if misses are removed. So, besides application to program optimization, this technique is an analysis tool that provides a means for evaluating the *true* impact of memory systems on overall program performance.

Program	Procedures	Original IPC	IPC after instrumentation	Potential improvement (speedup)
`171.SWIM`	`calc1`	0.41	1.78	4.35
	`calc2`	0.67	1.95	2.91
	`calc3`	0.62	1.61	2.60
`172.MGRID`	`resid`	1.23	1.82	1.48
	`psinv`	1.64	2.00	1.22
`173.APPLU`	`jacld`	1.19	3.08	2.59
`183.EQUAKE`	`smvp`	0.39	1.48	3.81

To fully validate the fact that the instrumentation *only* affects memory behaviour and that the upper-bound can effectively be interpreted as a *memory performance upper-bound*, we have performed an additional experiment using a full processor simulator. Using *SimpleScalar* [5] we modelled a superscalar processor with similar characteristics as the Alpha EV67 (however this is not an accurate model of the EV67, we can only state that it has *roughly* the same characteristics). We modified the simulator so that the cache and the TLB are perfect, i.e., all memory requests hit in the first-level cache and the TLB. Then, we have run both the original and the instrumented SWIM code on this simulator. Since the memory system is perfect, if both programs differ only in terms of memory behaviour, their performance on the processor simulator with a perfect cache should be nearly identical. Results in the table below confirm that instrumentation barely affects the overall program behaviour. Finally, we have run the same programs on the same simulator but with a normal cache, and we can see that the difference between both programs becomes very high both in terms of IPC and cache/TLB miss ratio.

	Original program	Transformed program
Normal cache	2.42	3.02
Perfect cache and tlb	2.98	3.02

IPC (Instructions Per Cycle).

	Original program	Transformed program
Number of L1 accesses	295,705,805	288,213,871
L1 Miss ratio	7.22%	0.0%
Number of TLB accesses	295,705,805	295,738,993
TLB Miss ratio	0.2%	0.0%

Cache behaviour (normal cache).

3.3 Guiding Optimizations

We have examined one program, SWIM, in more details and attempted to optimize it by hand. We have applied a large array of transformations: blocking, padding, loop merging and forward substitution.The table below shows the execution time of the original program, the manually optimized program, and the upper-bound (counting only the instrumented procedures).

Original	Manually optimized	Upper-bound
106.31	69.30	29.61

Execution time in seconds.

These results show that additional improvements potentially exist, beyond what we have already achieved, though the main limitation of our technique is that we provide an *upper-bound*, not a *maximum*, so we do not know whether it is *possible* to achieve this performance threshold. However, the technique provides an estimate of the potential gain, and while a small potential gain clearly means it is not worth pursuing the optimization effort, a large potential gain is an additional motivation. For instance, using iterative compilation techniques we achieved a 40% improvement on SWIM and only 10% on MGRID, i.e., a trend similar to the potential improvements of SWIM and MGRID indicated above.
Note also that the current technique would fit nicely in an iterative compilation framework. The goal of iterative compilation is to explore a large optimization space of the application without knowledge of the target machine, and to find out the optimal optimization sequence within that space. In practice, however, since the number of transformations is potentially infinite, the transformation space is necessarily restricted [12]. Using the method described in this article we may target those sections of the application program which have the potential to be improved so as to reduce the transformation space to be searched. Thus our method both gives a memory performance upper-bound and information on whether an application has a memory problem or not since, for example, all memory accesses may be hidden by intensive computations on out-of-order execution processors.

4 Caveats

The techniques mentioned above raise several issues that can sometimes complicate their implementation.

- With the Compaq Alpha compiler, the loop index, in the above examples, is usually distinct from the array reference register index. If the loop index were the same as the array reference register index, we would need an additional register to perform the same transformation since it is not possible to set the loop iteration increment as 0. However, besides that difficulty, it is likely this technique can be easily ported to other environments, especially if we have access to the production compiler.
- The scope of the techniques is restricted to loops and array references. It still encompasses a large set of Fortran codes, including sparse codes with indirect array references (the increment of the index array would then be set to 0). However, if a loop body contains a branch instruction, especially a branch instruction that depends on array values, then the technique cannot be applied as is. One solution is to evaluate the probability distribution of the branch outcome and then modify the test so that the branch outcome is generated by a random variable with this probability instead of the original test. This solution can perform reasonably well but it is partly satisfactory since the instrumented code instructions are not strictly identical to those of the original code, as for the loop bodies without branch instructions. Fortunately, a significant share of program loops in Fortran programs do not contain conditional branch instructions that depend on a value computed within the loop body.
- Superscalar processor architectures can include load/store queues that are designed to avoid consecutive accesses to identical memory addresses. For instance if a load to address `A` is issued after a store to `A` was performed and the store instruction is still in the queue, the load can directly access the data and avoid a memory reference. Our technique would potentially increase the number of such bypasses since, in a loop, an array reference is replaced by a reference to a constant so that the corresponding address is referenced many times. As a consequence, with such queues, our technique would provide an optimistic upper-bound. However, another related property may partially compensate this bias: in the instrumented programs there are many more load/store dependencies than in the original program since the overall number of addresses used is much smaller but the number of load/store instructions is the same. Such dependencies can degrade the exploitation of ILP in which case the upper-bound would be less optimistic than initially thought. Both effects must be investigated and evaluated in more details.

5 Related Work

There are several possible approaches to the problem of defining a program memory performance upper bound.
First, several studies [1, 4, 6] rely on Belady's MIN algorithm [3] to find the best possible memory behaviour but these studies only target a single memory level and the associated architecture is capable of selectively load, place and discard words and thus does not correspond to a real-life architecture.
The second most frequent approach is based on simulation. There is a very large amount of effort on simulation technology in the micro-architecture community [13, 5], including in the MHAOTEU project where we have developed the cache profiler and the cache debugger [18]. In [14], Martonosi et al. attempt to speed up cache simulation using trace sampling and achieve reasonably accurate results. However, simulation only provides a restricted view of the whole system performance, and it often ignores the impact of the operating system. Besides, much information on the system architecture is not available.

6 Conclusions and Future work

We have developed a technique for quickly evaluating the execution time of a program assuming most cache misses have been removed. The execution time upper bound is fairly accurate as the program is not

modified and is actually run on the machine studied. Thus all system and architecture artefacts are taken into account. The technique is way faster than simulation since the instrumented program execution time is at most double the execution time of the original program versus a 500+ slowdown for simulation techniques. Our goal is to have this technique implemented in a production compiler to assist users who whish to evaluate the impact of cache performance on their program execution time and to evaluate the progress achieved at each step of an optimization process. The main limitation is the scope of the techniques (loops; loops with conditional branches are more difficult to handle).
While this technique provides a program performance upper bound (a cache/TLB miss lower bound) it does not define whether or not this upper bound can be achieved although it allows us to spot the parts of the application which have memory problems and which are candidates for memory optimizations. Future research will focus on trying to define tighter upper bounds that are close to what can be effectively achieved through classic program transformation techniques. We can also use results from this research to narrow down the transformation space for iterative compilation without knowledge of the target machine which is particularly important in the presence of rapidly evolving hardware.

References

[1] Santosh G. Abraham and Rabin A. Sugumar. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, 1993.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, , and William E. Weihl. Continuous profiling: Where have all the cycles gone? *IEEE Transactions on Computer Systems*, 15(4):357–390, November 1997.

[3] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] D. Burger, A. Kägi, and J. R. Goodman. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd ACM International Symposium on Computer Architecture*, Philadelphia, May 1996.

[5] D.C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[6] Douglas C. Burger, Alan Kagi, and James R. Goodman. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report 1261, University of Wisconsin, Madison, April 1996.

[7] Stephanie Coleman and Kathryn S. McKinley. The Tile Size Selection Using Cache Organization and Data Layout. In *ACM SIGPLAN '95 Conference on Programming Languages Design and Implementation*, 1995.

[8] J. Dean, J. Hicks, Waldspurger, Weihl, and Chrysos. Profileme: Hardware support for inst.-level profiling on out-of-order. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, North Carolina, November, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[9] Rajagopalan Desikan, Doug Burger, and Stephen W Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Gteborg, Sweden, Jun 2001.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Edition.* Morgan Kaufmann Publishers Inc., 1996.

[11] R. E. Kessler, E. J. McLellan, and D. A. Webb. The alpha 21264 microprocessor architecture. In *Proceedings of International Conference on Computer Design*, Austin, Texas, December 1998.

[12] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In IEEE Press, editor, *Proceedings of PACT'2000, Parallel Architectures and Compiler Technology*, October 2000.

[13] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer, vol. 27, no. 10,pp. 15-26*, October 1994.

[14] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 248–259, Santa Clara, CA, May 1993. ACM SIGARCH.

[15] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, October 1996.

[16] Gabriel Rivera and Chau-Wen Tseng. A comparison of compiler tiling algorithms. In *Computational Complexity*, pages 168–182, 1999.

[17] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[18] Eric van der Deijl, Gerco Kanbier, Olivier Temam, and Elena D. Granston. A cache visualization tool. *IEEE Computer*, 30(7):71–78, July 1997.

[19] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.