Efficient detection of communication in multi-cores

Andreas Sandberg
Uppsala University
Department of Information Technology
Box 337, SE-751 05 Uppsala, Sweden
andreas.sandberg@it.uu.se

Stefanos Kaxiras
University of Patras
ECE Building
Rion Campus, Patras 26500, Greece
kaxiras@ece.upatras.gr

Abstract

Several methods have been proposed to model communication in systems with coherent caches, e.g. multi-cores, however they usually incur a large overhead on the application being analyzed. In this work we describe a low-overhead statistical communication model that is driven by a sparse sample of the memory accesses in the target application. Our model allows detection of hot-spots where coherence communication occurs between different threads in an application. Preliminary results suggest that we are able to detect most of the communication hot-spots in real applications with lower overhead than previously proposed models.

1. Introduction

The introduction of multi-cores is moving parallel programming from the domain of experts to more novice programmers. As more and more programmers start to exploit the parallelism offered by modern hardware, the need to understand the behavior of parallel applications increases.

Today we have a fairly good understanding of how to analyze and optimize serial code in respect to cache behavior. The most common approach is to use simulation, either on-line or off-line. An on-line simulator is attached to the target application and simulates the memory system as the application runs, whereas an off-line simulator is run from a stored memory access trace. There are several simulation based cache analysis tools, for example Cachegrind in the Valgrind[5] tool suite. Unfortunately, most such tools tend to suffer from a large overhead; slowdowns of several orders of magnitude are not uncommon.

Another drawback of on-line simulators is generally the lack of flexibility. Once the analysis is done, there is no way to change any of the cache parameters to simulate another cache level or processor. A way to increase flexibility would be to use an off-line simulator and only create a memory access trace on-line. This would indeed be the most flexible solution since all data relevant to a cache simulator exists in the trace file, however the size of the such trace files is usually prohibitively large.

To reduce overhead and improve flexibility some researchers have turned to statistical models. One such method, StatCache[2], uses a memory access sample to estimate the cache behavior of an application. Using a sparse sample instead of a full memory trace has the benefit of reducing the on-line overhead[1] and at the same time reducing the amount of data needed for the analysis. Furthermore, StatCache allows the user to change the cache size in the off-line phase of the algorithm. The authors of the original StatCache model later extended it to StatCache MP[3] which handles multiple processors with coherent caches.

The original StatCache algorithm uses a memory access sample consisting of pairs of accesses to the same cache line, for each such reuse pair the number of memory accesses to other cache lines is measured. In order to detect invalidations StatCache MP modifies the original sampling algorithm. The new sampling algorithm requires both accesses in a pair to come from the same thread. In addition to the memory access counter StatCache MP maintains a list of foreign threads writing to the cache line.

One of the drawbacks of the StatCache MP approach is that it requires a more complex sampling policy than the initial StatCache implementation. The stricter termination condition for access pairs causes more long-running pairs; this together with the additional complexity in the sampling application is likely to have a negative impact on performance.

We present an algorithm that detects communication sites in full-scale applications using a sampling policy similar, but somewhat simpler, to the one used in the original StatCache algorithm.

[1] The authors of the StatCache algorithm report overheads as low as 40% in average.

2. The communication model

Our communication model can be divided into two distinct parts; data acquisition, or *sampling*, and *data analysis*. Sampling is performed on-line by an application, the *sampler*, which monitors the target application's memory accesses. The output from the sampling phase is known as a *memory access sample*. The analysis phase uses the memory access sample to predict where communication takes place in the target application.

2.1. Sampling

Our communication model uses a sample of the target application's memory accesses. A memory access sample consists of a set of pairs of accesses, *reuse pairs*, to the same cache line. Each sampled memory access is annotated with its instruction address (*iaddr*), thread number and access type.

The first access in a reuse pair, *PC1*, is chosen randomly by the sampler. The second access, *PC2*, is the first access, from any thread, after PC1 that reuses the cache line accessed by PC1.

The sampler starts to monitor a new memory access on average every N memory accesses, where N is known as the *sample period*.

2.2. Data analysis

Using the memory access sample, we build a *weighted directed graph* of memory accesses. We start by building a map between thread numbers and caches; we assume a static mapping that is valid throughout the whole execution of the target application. Each vertex in the graph then corresponds to one unique iaddr-cache pair in the memory access sample. The edges in the graph represent sampled PC1-PC2 connections, with the weight of an edge being the number of times a particular pair has been sampled.

We assume that every branch in the application can be modeled as an independent random event and that program behavior is somewhat uniform throughout the execution. Using these assumptions we calculate the probability for a transition along an edge in the graph. Consider the vertex v with the adjacent vertex w, and let $W(v \rightarrow w)$ be the weight of an edge and let $E_{out}(v)$ be the set of all edges originating from v. The probability for the transition along the edge $v \rightarrow w$, $P(v \rightarrow w)$, can then be calculated.

$$P(v \rightarrow w) = \frac{W(v \rightarrow w)}{\sum_{e \in E_{out}(v)} W(e)} \tag{1}$$

The probability $P(v \rightarrow w_a)$ should be interpreted as the answer to the question "Given that an instruction v has just executed, what is the probability that the next instruction is w?".

Using the equation above we construct the cache state function, that is, the probability that a piece of data is available in a local cache at a given point in the program. We actually compute two functions, the *incoming state* and the *outgoing state*, i.e. the state before the instruction executes and the state after the instruction executes respectively. We denote the incoming probability for a cache line to be valid in cache c at vertex v by $P_c^I(v)$, the outgoing probability is denoted by $P_c^O(v)$, as described in Equation 2 and Equation 3 respectively where $C(v)$ is the cache that v is executing in.

$$P_c^I(v) = \sum_w P_c^O(w) P(w \rightarrow v) \tag{2}$$

$$P_c^O(v) = \begin{cases} 1 & C(v) = c \\ 0 & C(v) \neq c \wedge \\ & v \text{ is a write} \\ P_c^I(v) & \text{otherwise} \end{cases} \tag{3}$$

Equation 3 transforms the incoming cache probability into an outgoing cache probability. For an instruction v executing on a core with the local cache c, the outgoing probability is always 1, i.e. reading from memory installs the cache line in the local cache. In case the probability is not calculated for the local cache, the outgoing probability is always 0 if v is a write, a read on the other hand only propagates the incoming state.

Note that Equation 2 and Equation 3 are dependent and requires solving of a linear equation system. In all practical cases we have encountered so far, the system has been satisfied or over-satisfied and thus solvable.

Using the equations defined above we can compute the probability of a miss, $M(v)$, as the complement probability of a data item being in the local cache before an instruction, v, executes.

$$M(v) = 1 - P_{C(v)}^I(v) \tag{4}$$

The probability of sending an invalidating to foreign cache, $I_c(v)$, can also be expressed using the cache state functions.

$$I_c(v) = \begin{cases} P_c^I(v) - P_c^O(v) & c \neq C(v) \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

To calculate the expected number of misses for a specific instruction we need to know the number of times that instruction has been executed. This is calculated by multiplying the sample period, N, with the total weight of all edges originating from the vertex.

$$N(v) = \sum_{e \in E_{out}(v)} W(e) N \tag{6}$$

Benchmark	Accesses	Sample pairs 100	1000
BT	260 000 000	2 623 000	264 000
GS	360 000 000	3 645 000	366 000
Raytrace	120 000 000	1 209 000	121 000

Table 1: Number of accesses in the traces and the number of access pairs sampled from the traces using a sample period of 100 and 1000.

We then utilize the miss probability, $M(v)$, to calculate the expected number of misses, $N_M(v)$, for an instruction.

$$N_M(v) = N(v)M(v) \quad (7)$$
$$N_U(v) = N(v)U(v) \quad (8)$$

3. Preliminary evaluation

To evaluate the algorithm we counted the number of invalidation misses in the target applications using a trace driven cache simulator. We used the following benchmarks in the evaluation:

BT Implicit solver for 3-dimensional Navier-Stokes equation from the OpenMP implementation[4] of the NAS Parallel Benchmarks[1]. We used the class S version of the benchmark.

GS A naïve parallel implementation of a Gauss-Seidel smoother working on a 256×256 matrix of doubles. The algorithm was parallelized such that each core works on a set of columns and communication only occurs along the borders of a column.

Raytrace Ray-tracer from the SPLASH-2 benchmark suite[6]. We used the classical teapot scene included with the benchmark as input data.

We compiled all the benchmarks using GCC version 4.2.1 for x86-64 with `-O2` optimizations. All of the benchmarks were run with 2 worker threads.

The model was driven by a sparse sample taken from the trace used to drive the simulator, see Table 1 for a summary of the number of accesses in the trace and the number of access pairs in the sample files. We simulated a reference system with two private 4MB caches and 64B cache lines. To reduce noise in the model, we didn't include predictions for instructions having fewer than 10 sampled accesses or were based on less than 5 sampled accesses per vertex in average. We classified instructions having more than 50 000 invalidation misses as hot-spots.

Comparing the results from the simulator and communication in Table 2, we see that using a couple of million of access pairs we find most of the hot-spots in all of the benchmarks. Using a few hundred thousand access pairs we still detect all of the hot-spots in Raytrace and more than half of the hot-spots in GS, BT however proves to be much more elusive and we only detected a tenth of the actual hot-spots.

Benchmark	Simulator	Model	Overlap
BT	13	30	3
GS	5	6	3
Raytrace	6	12	6

(a) Sample period 1000

Benchmark	Simulator	Model	Overlap
BT	13	39	12
GS	5	8	5
Raytrace	6	11	6

(b) Sample period 100

Table 2: Breakdown of communication hot-spots detected in the benchmarks. The *simulator* column is the number of hot-spots that were detected by the simulator and the *model* column corresponds to the number of hot-spots detected by our model. The actual overlap between the two is in the *overlap* column.

In all of the benchmarks our model detected additional communication hot-spots that weren't detected by the simulator. We believe that most of these were actual communication sites, but with a slightly overestimated communication rate.

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, et al. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991.

[2] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. *SIGMETRICS Perform. Eval. Rev.*, 33(1):169–180, 2005.

[3] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–99, 2006.

[4] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. *NASA Ames Research Center, editor, Technical Report NAS-99-01*, 1999.

[5] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, UK, 2004.

[6] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodologicalconsiderations. *Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on*, pages 24–36, 1995.