Compiler Hints for Co-accessed Variables

Group
Valerie Choung (vchoung@andrew.cmu.edu)
Daniel Ramos (drramos@andrew.cmu.edu)
(and Edward Chen (ejchen) working on an overlapping project)

URL
http://nicebowlofsoup.com/coaccess

Project Description
In our project we will explore how object or variable *co-access* information can be useful in optimizing lower-level software stack components.

What is a Co-access?
First, we must consider what a co-access even is. On a high level, we say two objects or variables are co-accessed if an application accesses those variables within a short time duration. This is a rough description of a co-access, and part of our project will involve determining what a useful specific definition of a co-access should be. For example, the simplest definition of a co-access could be a pair of addresses that are each dereferenced within *N* instructions of the other with no exceptions.
Another aspect of co-accesses to consider is whether a co-access should be a uni- or bi-directional relation: if an access to variable *a* is always followed by an access to *b*, but an access to *a* doesn't always precede an access to *b*, then we need to choose whether we still call the pair *(a, b)* a co-access. This problem can be partially resolved by giving each co-access a notion of *strength*. A very strong co-access could be one where two variables are *always* accessed together and their accesses are very close and very frequent. A weaker co-access would be one where two variables are *mostly* accessed together, and their accesses are *somewhat* frequent.

Hotness-based Heap Data Placement
For our project, we will examine an optimization that would benefit from having co-access data:
We observe that strongly co-accessed variables share the same lifetimes and the same hotness. Using this insight, we expect that allocating strongly co-accessed variables contiguously in memory could benefit performance. In particular, small-enough variables could fit in a single cache line, reducing the number of cache misses. Additionally, less-hot variables would be flushed from the cache together, while hot variables stay in the cache together. Overall, this results in less data movement.

This can be done by replacing individual calls to *malloc* for co-accessed variables with a single call to *malloc* that would allocate enough memory for the entire clique of strongly co-accessed variables. For example, if *a* and *b* are strongly co-accessed with separate calls to *malloc*, we would replace the calls with a single large *malloc* to obtain a pointer *p*. Every time *a* is accessed, we dereference $p + n$, where n depends on the size of *a*. This approach has the nice benefit of reducing calls to *malloc* as well, which should additionally improve program performance.

In principle, this sounds fairly simple, but if these *malloc* calls appear in a loop or if the allocation sizes

are not known until runtime, we will need to employ a JIT-style dynamic analysis to coalesce the calls to *malloc*. We illustrate two scenarios below:

(1) Sizes unknown at runtime:

```
a = ?
p = malloc(a);
b = ?
q = malloc(b);
```

Is transformed into

```
a = ?
b = ?
p = malloc(a + b); <-- need to generate this malloc on-the-fly
q = (char *)p + a;
```

(2) Loop

```
loop {
  p = malloc(5);
}
```

Suppose all the *p*s are co-accessed. If the loop runs *N* times but *N* is unknown, we can replace the *i*th call to *malloc* (if *i* is a power of 2) with *malloc(2^(i-1) * 5)* here for worst case ½ utility with *lg N + 1* calls to *malloc*.

Goals

75%: Determine (statically) co-accessed variables and handle non-looped calls to *malloc* with unknown sizes, as shown in Scenario 1 above.

100%: Same as 75%, but also be able to dynamically identify if objects allocated within a loop are co-accessed (Scenario 2).

125%: Same as 100%, but also perform the transformation described above for Scenario 2.

Plan of Attack

We will pair program the more complex logic, and split the small tasks for this project.

Week of 3/28: Simple static co-access analysis with some notion of strength.
Week of 4/4: Coalesce calls to *malloc* with sizes known at compile time.
Week of 4/11: Dependency analysis for *malloc* sizes not known at compile time and move size computations around accordingly.

Week of 4/18: Design co-access analysis for variables allocated within a loop.
Week of 4/25: Implement co-access analysis for variables allocated within a loop.

Milestone
See Plan of Attack week of 4/11 (above).

Literature Search
We've done a fair amount of literature search so far:

BEL, O., CHANG, K., TALLENT, N. R., DUELLMANN, D., MILLER, E. L., NAWAB, F., AND LONG, D. D. E. Geomancy: Automated performance enhancement through data layout optimization. In 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2020), pp. 119–120.

RAMAN, E., HUNDT, R., AND MANNARSWAMY, S. Structure layout optimization for multithreaded programs. In International Symposium on Code Generation and Optimization (CGO'07) (2007), pp. 271–282.

Co-access data does not seem to be directly used in many modern optimizations.

Resources
No need for anything else. We are using the same VM environment provided for our assignments.

Getting Started
We have some rudimentary code for statically identifying co-accesses. We also have a framework for instrumenting code, which will be useful for dynamic analysis as well and inserting calls to *malloc*.