Identifying Critical Loads in Real Programs for Decoupled VSM Systems

He Zhu and Ian Watson

Manchester University

Abstract. We present an interprocedural flow-sensitive analysis to identify critical load instructions in real C programs for Decoupled Virtual Shared Memory (DVSM) systems. The analysis consists of the pointer range analysis, the critical value analysis and the identifying steps. It is implemented based on the SUIF compiler and tested using programs from the SPLASH-2 suite. The results show that the method is safe.

1 Introduction

The idea of decoupling is suggested to split the address calculation and the normal computation streams to reduce memory latency [3]. Simulations on DVSM architectures with a dual CPU processor structure have shown optimistic results [4]. In each processor node, both the execution CPU (CPU B) and the prefetch CPU (CPU A) run identical codes. CPU B performs all operations as usual, and it will wait if a page is not locally available. CPU A does the same, but it will not wait even if a page fault happens, instead it issues a prefetch operation, and gets a special *dirty value* immediately, and then the execution continues. A performance benefit is awarded because CPU A can go ahead of CPU B and prefetch remote pages, hiding remote request latency. But loss of decoupling events may arise, because of the introduction of dirty values. For example, in Fig. 1, when accessing a[b[i]] and b[i] is dirty, CPU A may fail to prefetch the correct element of a. A special critical load instruction (*LDC*) is introduced to force CPU A wait whenever a dirty value is not allowed. Generally, we use *critical* in this paper to refer to something that can not tolerate dirty values.

We suggest an interprocedural flow-sensitive analysis based on a Full Data-Flow Graph (FDFG) with the support of the SUIF system [1] to detect critical load instructions automatically.

2 A Solution to Identify Critical Load Instructions

Given a load instruction, if at least one of its instances accesses a global location, and the loaded value is critical, then the load instruction is an LDC. It is desired that CPU A will proceed along exactly the same control flow and access exactly the same memory locations as CPU B. A critical value means a value which is

```
int x, a[100], b[100];
proc1(){
  int i;
  if (x)       /* load to x should be critical */
    x = x + a[b[i]]  /* s1: load to x is not necessarily critical*/
                     /*     but load to b[i] should be critical */
  else
    x = x + 1;       /* s2: load to x is not necessarily critical */
  proc2(x);          /* load to x should be critical, because the
                        value is  critical inside proc2 */
}
proc2(int arg1){
  if (arg1) x = 0;
}
```

Figure 1. Some Critical Loads in an Example Program

used as an operand address or as a transfer condition. In Fig. 1, there are three LDCs.

Dirty values are only introduced by a global load on CPU A with a page fault. They exist in registers and local memory locations. Global memory locations are isolated from dirty values by the system which assumes certain memory coherency. A critical global load always provides a clean value, and a normal global load can load a dirty value if a page fault occurs. A dirty value can propagate through registers and local memory.

When tracing dirty values, there are two methods of finding critical loads. A *Register-dirty method*, simple but conservative, traces only intermediate values in registers and some temporary memory locations. It is assumed that any dirty value is not allowed to propagate into any other local memory locations. A *Local-dirty method*, powerful but complicated, releases the above constraint by allowing any local memory location to hold a dirty value conditionally, only if this dirty value is not used in any critical operation. The load which introduces such a dirty value need not be critical.

To apply the local-dirty method, we have designed a *Full Data Flow Graph* (FDFG) which is based on an Interprocedural Control Flow Graph(ICFG). In an FDFG, for each data item, a *data node* is explicitly created. Interprocedural dependence is represented by edges which connect interprocedural data nodes between *call sites* and *callees*.

The critical load analysis operates on the FDFG of a program. It first analyzes every pointer to compute their pointer ranges. Then a critical value analysis is applied to trace all critical values. Finally, all critical loads are picked out by checking operation nodes in the FDFG.

Pointer Range Analysis The pointer range analysis is intended to decide the memory ranges referenced by each pointer. Unlike the pointer analysis in [5] which tries to compute precise *points-to* information for each pointer in the program, in our analysis only two memory ranges are distinguished from each other. They are the *shared memory space (global space)* and the *private memory space (local space)*. It consists of the initialisation and the propagation steps. The steps use marks to distinguish address ranges. The propagation step tries to propagate these marks through the FDFG according to the *propagation rules*, which are developed by control flow and operation nodes.

Critical Value Analysis This analysis is for identifying all the critical values in the program by marking critical data nodes. It consists of setting address marks, initial marking for critical nodes and back-tracing critical values steps. Starting with a critical value, we can trace backwards along the definition-use links from a critical value to identify all the other critical values and make sure no dirty values could propagate into any critical location.

Finding Critical Loads in Program The final step to identify critical loads is just to check all load operations in the program to decide if they are critical loads. Then set a mark for each critical operation node. The marked FDFG can be transferred back to a C program with the mark information reflected in the source lines. A program transformation will employ this information to get a transformed source program.

3 Experiments

We have implemented a tool, known as Critical Analysis and Program Transformation (CAPT), based on the SUIF compiler, to identify LDCs. Examples with complicated interprocedural relations are successfully processed with optimistic results. Fig. 2 gives the result for the code in Fig. 1 produced by CAPT, where LDCs are marked by special *critical_load* functions.

Some real programs and applications from the SPLASH-2 suite [2] have been tested using CAPT. The results(see Fig. 1 show that LDCs in a program usually will represent no more than 10% of the total memory access instructions.

Program	water	barnes	raytrace	fmm	radiosity	ocean
lines	1776	2303	10022	3847	22118	4712
critical loads	435	426	787	750	1345	1622
accesses	4857	5199	14133	7362	25462	7852

Table 1. Statistics on Experimenting Programs in SPLASH-2 using CAPT

```
int x, a[100], b[100];
proc1(){
  int i;
  if (critical_load(x))
    x = x + a[critical_load(b[i])]
  else
    x = x + 1;
  proc2(critical_load(x));
}
proc2(int arg1){
  if (arg1) x = 0;
}
```

Figure 2. Identified Critical Loads

4 Conclusion

The prefetch benefit exists if there is any global load ,which results in a page fault, but is not critical. The solution has a good performance because, in most cases, it can detect load instructions which need not be LDCs. It is safe because it will not fail to mark any LDC. CAPT is implemented using C++, and the execution efficiency has not been emphasised in our research.

We have not found any other papers dealing with the LDC problem.

Our experiments show that a solution by data-flow analysis for the problem is possible. But more work is needed to make it efficient. Our analysis, although not as aggressive as some other pointer analyses, is not limited to distinguishing between two memory ranges but could be improved for other pointer analysis applications.

References

1. S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An overview of a compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing.* The Stanford SUIF Compiler Group, August 93.
2. J.P. Singh, W. Weber, and A. Gupta. Splash:. Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1), March 1992.
3. J.E. Smith. Decoupled access/execute computer architecture. In *ACM Transactions on Computer Systems, Vol.2, No.4*, pages 289–308, November 1984.
4. I. Watson and A. Rawsthorne. Decoupled pre-fetching for distributed shared memory. In *Proceedings of the 28th HICSS, Vol 1*, pages 252–261, 1995.
5. Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In David W. Wall, editor, *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, volume 30(6) of *ACM SIGPLAN Notices*, pages 1–12, New York, NY, USA, June 1995. ACM Press.