Strategies for Cache and Local Memory Management by Global Program Transformation

By

Dennis Gannon, William Jalby and Kyle Gallivan

TECHNICAL REPORT NO. 228

Strategies for Cache and Local Memory Management By Global Program Transformation

by

Dennis Gannon, William Jalby and Kyle Gallivan

September, 1987

This report is based on work supported in part by the Air Force Office of Scientific Research under Grant No. AFOSR-86-0147.

Strategies for Cache and Local Memory Management by Global Program Transformation

Dennis Gannon

Dept. of Computer Science, Indiana University and
CSRD Univ. of Illinois, Urbana-Champaign.

William Jalby

INRIA and CSRD Univ. of Illinois, Urbana-Champaign

Kyle Gallivan

CSRD Univ. of Illinois, Urbana-Champaign.

ABSTRACT

In this paper we describe a method for using data dependence analysis to estimate cache and local memory demand in highly iterative scientific codes. The estimates take the form of a family of "reference" windows for each variable that reflects the current set of elements that should be kept in cache. It is shown that, in important special cases, we can estimate the size of the window and predict a lower bound on the number of cache hits. If the machine has local memory or cache that can be managed by the compiler, these estimates can be used to guide the management of this resource. It is also shown that these estimates can be used to guide program transformations in an attempt to optimize cache performance.

September 11, 1987

Strategies for Cache and Local Memory Management by Global Program Transformation

Dennis Gannon

Dept. of Computer Science, Indiana University and
CSRD Univ. of Illinois, Urbana-Champaign.

William Jalby

INRIA and CSRD Univ. of Illinois, Urbana-Champaign

Kyle Gallivan

CSRD Univ. of Illinois, Urbana-Champaign.

1. Introduction

Perhaps the most critical feature in the design of a shared memory parallel processor is the organization and the performance of the memory system. Generally, the shared memory is implemented as a set of independent modules (which, in turn, may be interleaved) connected to the processors via either a bus or a switch network. This organization exhibits several potential performance problems. First because of the long journey each memory reference must traverse in going to and from this shared resource the latency to access a data may be pretty high; this effect ends up to be very important in systems with large number of processors where the number of network stages that each memory request has to go through, grows as $\log_2(p)$ where p is number of processors (due to the number of stages of the network). Second due to the contention both at the network level and the memory level (routing and memory banks conflicts), the practical bandwidth (and also the latency) can be severely degraded. A good example of such phenomenon is "hot spot" contention [PhNo85]. To overcome these problems, the key idea is to use a hierarchical memory system, technique already proven efficient on sequential computers for speeding up memory access.

In such a system the memory is organized in several levels which might be fully shared (each processor may access the whole level), partially shared (the processors "inside a cluster" share the access to a given level) or fully private (each processor has its own level, access of which is restricted to itself; therefore access do not suffer from going through the communication medium and from interfering with the other processors). The transfers between these levels are either entirely hardware managed (such as with a cache where the user has no explicit control on the loading and unloading strategies) or fully software managed (such as registers where the user, or preferably, the compiler explicitly moves data between levels). For example the ALLIANT FX8 uses two levels of shared memory (trading size for speed): the main memory is connected via a bus to a high bandwidth cache which is turn shared by the processors through a crossbar. Access to vectors from the cache is 2 to 3 times faster than accessing from memory. Additionally, each processor has its own instruction cache and its own set of vector registers. In other machines only the main

memory level is fully shared and each processor has a private data cache, such as with the Sequent Balance and Encore Multimax, or a local memory which might be physically distinct from the main memory (CRAY2) or just a portion of the shared memory such as the IBM RP3 or BBN Butterfly. In this last case, the difference between local and shared requests are that the local ones do not go through the network. Finally, the Cedar system has perhaps one the most ambitious design of memory hierarchy. At the processor level, we have instruction caches and vector registers. The processors are then grouped into clusters which share a local cluster memory which is accessed by a shared cluster cache. Finally the clusters have access to the globally shared memory through the network. Additionally, each cluster has a prefetch unit controlled by software for prefetching data and therefore hiding the latency induced by the access to the global memory level.

However, the overall performance of all these hierarchical memory systems is highly dependent upon the address reference stream of the program (more precisely its locality). Several studies ([CJST85] [GaJaM87] [GaJa87]) have shown how algorithm reorganization may result in considerable performance improvement. It is crucial to notice that even in the case of hardware managed systems (ALLIANT FX8), reorganizing the program in order to make references to the same variable closer in time (or reducing the size of the working set) will speedup program execution [GaJaM87] [GaJa87]. Similar phenomena were already observed in studies considering the effect of program organization on paged memory behavior [KaMK69] and our approach can be considered following in the spirit of [AsKL81].

In this paper we consider the problem of automating the process of transforming programs to optimize the utilization of the memory hierarchy. For sake of simplicity, in our first approach we will assume that the transfers between levels are completely under software control. In that case, locality optimization is a 2-level process: first for a given program, one must solve the allocation problem (which data is to be kept in cache and for how long), and then restructure the program to minimize the number of data transfers. In fact, our basic strategy may still be applied to hardware managed systems. This is because reducing the number of transfers between levels is a dual problem to maximizing the reuse of data (optimizing "hit ratio"). Our key idea in solving the allocation problem is to consider it at a macroscopic level (loop level and section of arrays) rather at the microscopic level (machine level instruction and individual array elements). Using the fact that for scientific codes, most of the CPU time is spent in loop-like structure execution, we will globally study the interaction between two statements in a loop, by analyzing the sets of all the addresses referenced by each of them during the whole loop execution.

First we show that the theory of data dependence analysis used in automatic vectorizing compilers can be extended so that a more refined algebraic structure can be given to a class of data dependences associated with array index expressions that are common in scientific code. We call this class of dependences "uniformly generated". Next we associate a "reference window" with each data dependence. The reference window of a data dependence between two statements describes the set of elements (section of the array) that must be kept in the fast memory level to make sure that any data referenced by both statements will stay in the fast level as long as both statements continue to be executed and continue to reference that data item. More generally, we