LODS: Locality-Oriented Dynamic Scheduling for On-Chip Multiprocessors*

Mahmut Kandemir
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
kandemir@cse.psu.edu

ABSTRACT

Current multiprocessor SoC applications like network protocol codes, multimedia processing and base-band telecom circuits have tight time-to-market and performance constraints, which require an efficient design cycle. Consequently, automated techniques such as those oriented towards exploiting data locality are critical. In this paper, we demonstrate that existing loop scheduling techniques provide performance improvements even on on-chip multiprocessors, but they fall short of generating the best results since they do not take data locality into account as an explicit optimization parameter. Based on this observation, we propose a data locality-oriented loop-scheduling algorithm. The idea is to assign loop iterations to processors in such a fashion that each processor makes maximum reuse of the data it accesses.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *compilers, optimization.*

General Terms

Performance, Languages.

Keywords

Data Locality, Parallelization, Loop Scheduling, Embedded System.

1. INTRODUCTION

Many computer architecture companies such as Sun Microsystems Inc., IBM, Compaq Computer Corp., and Hewlett-Packard Co. have already started to produce systems that take advantage of chip multiprocessing, a radical step forward from current systems that load up boxes with multiple discrete chip modules. A principal analyst at microprocessor consulting firm The Linley Group in Mountain View, California indicates, "The driving force here is, rather than create more complicated processors, why not just put two in the same module? Over time, operating systems will become multiprocessor-chip-savvy, but the programming hurdles will be difficult to overcome" [1]. One of the tools that can greatly help programmers in this direction is optimizing compilers. An optimizing compiler can automate several optimizations under a unified framework, which can help satisfy performance, power, and space requirements of today's applications.

While optimizing for data locality is important for all types of multiprocessor machines [3], the issue is much more critical for on-chip multiprocessors. This is because in these architectures off-chip accesses are much more costly as compared to on-chip communication (in terms of both performance and power). In addition, poor data locality can dramatically increase the number of costly off-chip accesses, which can in turn degrade the overall performance. Therefore, it is of extreme importance that application execution exhibits decent data locality – i.e., most data requests should be satisfied from on-chip caches/memories instead of off-chip memories.

One of the most important problems in executing loop-based applications in on-chip multiprocessors is to decide how loop iterations need to be distributed across available processors (this is called loop parallelization). Most of current loop scheduling techniques for multiprocessors do not take data locality explicitly into account. Rather, reliance is placed upon the observation that in most of loop-based applications, assigning a block of successive loop iterations to each processor leads to acceptable data cache behavior. While this may be true for some applications, in some other cases where data sharing is not uniform across processors, current loop scheduling techniques may not perform well. As a result, designing locality-conscious loop scheduling techniques is a promising research direction.

In this paper, we demonstrate that existing loop scheduling techniques (developed in the context of high-end large-scale parallel machines) provide performance improvements even on on-chip multiprocessors, but they fall short of generating the best results since they do not take data locality into account as an explicit optimization parameter. Based on this observation, we propose a data locality-oriented loop-scheduling algorithm. The idea is to assign loop iterations to processors in such a fashion that each processor makes maximum reuse of data. Our algorithm achieves this by (logically) dividing arrays into sections, and assigning all the loop iterations that manipulate the same array section to the same processor. It also maintains load balance by updating processor workloads at runtime as necessary. The rest of this paper is organized as follows. The next section gives background information on loop scheduling techniques. Section 3 presents our locality-oriented loop-scheduling strategy in detail. Section 4 concludes the paper by summarizing its major contributions.

* This research is supported in part by NSF Career Award #0093082, and a grant from Gigascale Systems Research Center.

DAC 2004, June 7–11, 2004, San Diego, California, USA

2. BACKGROUND ON LOOP SCHEDULING

Loop scheduling techniques can be broadly divided into two major groups: *static* and *dynamic* [8]. In static scheduling (also referred to as *pre-scheduling*), the loop iterations that will be executed by each processor at runtime are completely determined at compile time. In the most common used static scheduling technique, if there are N iterations to assign to P processors, $\lceil N/P \rceil$ consecutive iterations are assigned to each processor; this is referred to as *block scheduling* (Another method, called *cyclic scheduling*, assigns every P^{th} iteration to each processor – We have not considered cyclic scheduling here since our initial experiments with it showed very poor results due to poor data locality). In contrast, in dynamic scheduling (also called *self-scheduling* [7]), the loop iterations to be executed by a processor are decided at runtime. There is a clear tradeoff between these two rival techniques. While static scheduling techniques do not incur any runtime penalty (since all scheduling related decisions are taken at compile time), they also tend to cause load imbalance, particularly in cases where different loop iterations can be of different costs. This can happen, for example, when the bounds of an inner loop that resides within a parallel outer loop are dependent on the loop index of the latter, or when there is a conditionally executed statement within the loop body. Therefore, dynamic scheduling techniques may perform better from the viewpoint of inter-processor load balance. However, since loop iteration allocations are decided at runtime, they can cause significant runtime overhead.

Most of previously proposed loop scheduling techniques (see [8] for a discussion) target at reducing runtime overheads, reducing inter-processor synchronization, and/or improving load balance across processors. Recent trends in embedded systems world, however, clearly indicate that optimizing data locality is becoming very critical as many embedded systems are employing data caches. A data locality-unconscious loop scheduling can result in poor cache behavior since there is no guarantee that the iterations assigned to a processor reuse the same set of data [4]. A few prior efforts such as [5] that consider data locality during loop scheduling operate under the implicit assumption that nearby (successively executed) loop iterations make good reuse of data. While this may be true in some applications, there also exist many codes where reuse distance (i.e., the difference between the loop iterations that use the same data) is very high. In fact, in many loop nests encountered in practice, loop transformations may not be able to reduce reuse distances due to intrinsic data dependences in the loop nest.

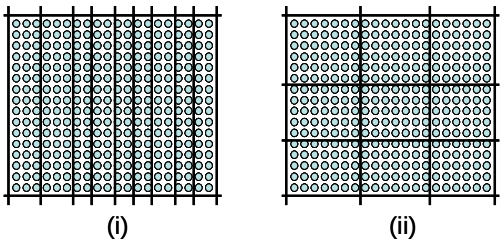

Figure 1. Two different divisions for a two-dimensional array when we have 9 processors. (i) Column-block and (ii) Block-block.

3. LOCALITY-ORIENTED DYNAMIC SCHEDULING (LODS)

3.1 Assumptions

We assume an on-chip multiprocessor, where each processor can execute independently from the others (it can execute either a different application than the others, or it can concurrently execute a portion of the same application). The compiler support required involves parallelizing a given loop across processors. That is, for each loop nest in the program to be executed, the compiler distributes iterations of the parallel loop (in the nest) across processors, and ensures that (by generating and inserting appropriate code at compile time) the processors synchronize at the end of each parallel loop. Since as far as the objectives of this paper is concerned it does not matter which loops are parallelized, we assume that our algorithm is already given this information (to be more specific, the compiler is tuned such that for each nest only the outermost loop without any data dependences has been parallelized). Our main target domain is array-intensive embedded applications. These applications usually consist of loop nests and multidimensional arrays. It is assumed that each processor has a cache memory, and therefore, ensuring data locality is critical.

It should be stressed that our work is fundamentally different from existing work on operating system (OS) level scheduling techniques for parallel architectures (e.g., [2]). In contrast to them, our scheduling is finer-granular (performed for a single application), and dictated by the compiler rather than the OS. Detailed comparison of (and interactions between) these two types of scheduling techniques is beyond the scope of this paper. The work presented in this paper is also different from energy-oriented scheduling techniques proposed by prior research [9] as our focus in this paper is on performance.

3.2 Our Approach

We propose a novel loop-scheduling technique, called LODS, which takes data locality (cache behavior) into account. Our approach, which can be best described as *hybrid* as it has characteristics of both static and dynamic scheduling, consists of two major steps:

(i) Loop iterations are grouped into *bins*, where each bin holds iterations that exhibit high degree of data reuse.

(ii) Each bin is originally assigned to a single processor, and is scheduled to be executed by it. When a processor finishes its allocation earlier than the others, it takes some loop iterations from another bin.

It should be observed that the second step above is similar to the one employed by Markatos et al in their work [5], and aims primarily at improving load balance across processors when it becomes an issue. The novel part of our approach though is the first step, which is detailed below.

Let us first consider a single array scenario; later we discuss how our approach extends to multiple arrays, which is obviously a more realistic case. Recall that our first goal is to divide loop iterations into bins (and we want each bin to contain iterations that access mostly the same set of data elements). Without loss of generality, let us assume that we have P processors in our chip multiprocessor. In the first step, we divide the array in question into P sections. Figure 1 illustrates two such divisions for a two-dimensional array

assuming that P=9. In the ideal case, we want each section to have roughly the same number of array elements.

In the first division (i), each section is given a set of consecutive columns (called the column-block division), and in the second one (ii) each section is a two-dimensional subblock (called the block-block division). In the second step, for each array section, we determine the set of iterations that access the elements in that section. In mathematical terms, we have:

$$I_s = \{ I \mid R(I) = d \ \wedge \ d \in D_s \},$$

where D_s is the s^{th} section of the array, and I_s is the set of iterations as defined above. Here, R represents a reference to the array in question, and R(I) represents the array element accessed by loop iteration I – i.e., the array element being accessed. It should be stressed that what set I_s contains is all the loop iterations that access (through reference R) only the elements in array section D_s. Note also that two such iterations in I_s can be far apart from each other in the original loop nest (our scheduling algorithm tries to assign them to the same processor to improve performance).

As a concrete example, if X[a+3,a+b-2] is a reference to an N-by-N array X within a nest that contains two loops (a is the outer loop index and b is the inner loop index), then if :

$$D_s = \{ d \mid d=(x_1 \ \ x_2) \ \wedge \ L_1 \le x_1 \le U_1 \ \wedge L_2 \le x_2 \le U_2 \ \},$$

we obtain:

$$I_s = \{ I \mid I=(a \ \ b) \ \wedge \ x_1 = a+3 \wedge x_2 = a+b-2 \wedge (x_1 \ \ x_2) \in D_s\}.$$

It should be noticed that D_s in this example represents array section $X[L_1..U_1, L_2..U_2]$, and that such a set can be written for each array section. I_s, on the other hand, represents the set of loop iterations (and only those loop iterations) that access D_s through reference X[a+3,a+b-2].

If there are multiple references to the array in question, we first determine a *global reference* that reflects the accesses via all references (to the array) combined. The idea can be best explained through an example. Suppose that we have two references, X[a,b+1] and X[a-1,b], and $2 \le a \le N$ and $1 \le b \le N-1$. Considering the first reference, one can see that the array region accessed via this reference is X[2..N, 2..N]. Similarly, when one considers the second reference, the accessed region is X[1..N-1, 1..N-1]. Consequently, considering both the references, one can determine the accessed region as X[1..N, 1..N], which can be represented by the global reference X[a,b] under $1 \le a \le N$ and $1 \le b \le N$. Our approach then proceeds with this new reference as described earlier. While in some cases a global reference may represent more elements than actually accessed (by the involved references combined), this does not present any correctness issue since we would not have any iteration points (in I_s) for such extraneous elements. Also, in some cases, when the I_s sets are determined based on the global reference concept, an iteration may need to be assigned to multiple bins. If this occurs, we assign it to the bin whose iteration accesses the array element in question using the left-hand-side reference (that is, such iterations are assigned to the bins considering the left-hand-side reference rather than the global reference).

In our approach, each I_s represents the contents of a bin. It should be noticed that since each I_s is determined based on an array section and each section may have a different number of array elements, the number of loop iterations assigned to a given bin may be different from the others. In fact, it is possible (though not very frequent) that some bins may be empty. In the third step, each bin is assigned to a different processor, and this concludes the static part of our loop-scheduling algorithm. It must be observed that since the iterations in I_s access only the elements of D_s, they exhibit very good data locality. As a result, by assigning these iterations (that is, the I_s set) to a processor, our approach can improve cache behavior dramatically.

The dynamic part of our algorithm tries to eliminate load imbalance when it occurs. Since, as noted above, each bin may have a different amount of workload, it is possible that some processors will finish their allocations earlier than the others (which means load imbalance). The dynamic part of our algorithm is responsible from detecting this situation when it occurs, and assigning some of the iterations to be executed to idle processors, in an attempt to balance the workload across them. While it is possible to select the processor from whom to take iterations randomly, in our approach this selection is also made through an analysis at compile time. Specifically, for each bin, we associate an *affinity index* (at compile time) – AI –, which points to the bin that exhibits good data reuse between them. Our algorithm achieves this by considering the references in the loop nest. Let us assume that X[a+3,a+b-2] is our left-hand-side reference and we have two right-hand-side references: X[a+b+1,b-4] and X[a,b]. Assume further that we have two array sections: D_{s1} and D_{s2}. We proceed as follows (for the sake of presentation clarity, we use the left hand side reference rather than the global reference to determine I_s). Let us assume that:

$$D_{s1} = \{ d \mid d=(x_1 \ \ x_2) \ \wedge \ L_1 \le x_1 \le U_1 \ \wedge L_2 \le x_2 \le U_2 \ \}.$$

We have:

$$I_{s1} = \{ I \mid I=(a \ \ b) \wedge x_1 = a+3 \wedge x_2 = a+b-2 \wedge (x_1 \ \ x_2) \in D_{s1}\}.$$

Then, the array elements accessed by this iteration set via the first right-hand-side reference can be written as:

$$RS(D_{s1},I_{s1},X[a+b+1,b-4]) = \{ d \mid d=(x_1,x_2) \ \wedge x_1 = a+b+1 \ \wedge \ x_2 = b-4 \ \wedge \ (a \ \ b) \in I_{s1} \}.$$

Similarly, the elements accessed by this I_s via the second right-hand-side reference can be expressed as:

$$RS(D_{s1},I_{s1},X[a,b]) = \{ d \mid d=(x_1,x_2) \wedge x_1=a \wedge x_2=b \wedge (a \ \ b) \in I_{s1} \}.$$

Based on these, the total set of array elements accessed by the iterations in I_s are:

$$T_{s1} = D_{s1} \cup \ RS(D_{s1},I_{s1},X[a+b+1,b-4]) \ \cup RS(D_{s1},I_{s1},X[a,b]).$$

In a similar fashion, for our second section (D_{s2}), we can determine:

$$T_{s2} = D_{s2} \cup \ RS(D_{s2},I_{s2},X[a+b+1,b-4]) \ \cup RS(D_{s2},I_{s2},X[a,b]).$$

Consequently, the affinity index (AI) between I_{s1} and I_{s2} is:

$$AI(I_{s1},I_{s2}) = | T_{s1} \cap T_{s2} |,$$

which is the number of common elements in T_{s1} and T_{s2}. Once these computations have been performed, we mark the bins in such a fashion that the bin that holds I_{s1} is associated with the bin that holds I_{s2} if and only if the affinity index between I_{s1} and I_{s2} is larger than that between I_{s1} and any other I_{s3}. This indicates that the sets of elements accessed by I_{s1} and I_{s2} have a large number of common elements. Therefore, if the processor that executes I_{s1} finishes early, it can borrow iterations from the processor that executes I_{s2}.

An important question to address here is how to calculate I_s sets at compile time. We propose to employ a polyhedral enumeration tool for this purpose. Specifically, we use Omega Library from the University of Maryland [6], which is a system for manipulating sets of affine constraints over integer variables. It enumerates sets constructed using Presburger formulas. Presburger formulas are those formulas that can be constructed by combining constraints on integer variables with the logical operations and the quantifiers such as $\forall$ and $\exists$. The constraints can be either equality constraints or inequality constraints. As an example, the description of I_s given above is a set of Presburger formulas, and thus its elements can be enumerated using the Omega Library.

Let us now consider the case with multiple arrays (which is usually the case in many embedded applications). It should be observed that what our approach for the one array case essentially does is to decide the bins for processors based of the division of the array in question. The main question that we need to answer when we move to the multiple array case is which array to use for determining the I_s sets. This is because each array can lead to a different bin contents (i.e., different I_s sets), which can in turn generate very different runtime behavior. Our solution to this problem is as follows. Since in a given loop nest there are usually very few arrays, we can try each array in turn (i.e., we try each array as a candidate array that can be used for determining I_s), and select the one that leads to the best overall execution behavior. Then, the question is how to decide (at compile time) whether one array will generate better execution behavior than another one.

In mathematical terms, let us assume that we have K different arrays accessed in a given nest: X_1, X_2, …, X_K. Without loss of generality, let us consider array X_j as the one to be checked to see whether it should be used for deciding the bin contents ($1 \leq j \leq K$). Based on the division of this array, we can determine the I_s sets as described earlier in the paper. Then, we compute:

$$RS(D_s,I_s,X_i) = \{ d \mid d=(x_1,x_2) \wedge [\exists R \text{ such that } R(I) = d \wedge I \in I_s]\}.$$

What this set contains is the set of elements accessed from array X_i by the iterations in I_s. Note that here R represents a reference to array X_i. Then, the total number of elements accessed by the iterations in I_s can be written as:

$$T(X_j) = \sum_{D_s} \sum_{i} | RS(D_s,I_s,X_i) |.$$

Note that, in calculating $T(X_j)$, we consider each section of each array in the loop nest being optimized. While this is a computationally intensive process in general, we can make two arguments in favor of it. First, since the array sections are expected to be big, the first summation above does not iterate too many times. Second, all these calculations are done at compile time, and in embedded computing, we can afford to longer compilation times as the code quality is of extreme importance. We compute $T(X_j)$ for each array X_j ($1 \leq j \leq K$), and select the one that gives the minimum $T(X_j)$ – i.e., the one the accesses the smallest amount of data.

3.3 Algorithm

Figure 2 gives the static portion of our scheduling algorithm in a pseudo-code format. In Steps 1 and 2, we identify the array that will help us determine the I_s sets. In Step 3, the array selected is logically divided into P sections. In step 4, we assign iterations to bins, and in the last two steps we determine (for each bin) the bin to be used in case the corresponding processor finishes its iterations early.

```
For each loop nest:
  1. For each array Xj:
    1.1. Compute T(Xj)
  2. Identify the array Xj with minimum T(Xj)
  3. Divide array Xj into P sections
  4. For each section Ds:
    4.1. Compute the global reference
    4.2. Compute Is
    4.3. Assign each Is to an unused bin
  5. For each (Is1, Is2) pair:
    5.1. Compute affinity index AI(Is1, Is2)
  6. Associate a next bin for each bin
```

Figure 2. The sketch of the static portion of our locality-oriented scheduling algorithm.

4. CONCLUSIONS

By putting multiple CPUs on a single piece of silicon, embedded system engineers can take advantage of shorter distances and faster bus speeds when shuttling data between the CPU cores. The net performance result is the ability to process more data per second and reduced power consumption. In this paper, we propose a new loop scheduling strategy for array-intensive embedded applications that execute on on-chip multiprocessors. The distinguishing characteristic of our approach is that it assigns loop iterations to processors assuming a logical division of arrays; this helps improve data locality.

5. REFERENCES

[1] http://www.itworld.com/Comp/1092/ CWSTO54343/

[2] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[3] M. S. Lam. Locality Optimizations for Parallel Machines. Parallel Processing: CONPAR 94 - VAPP VI. *Third Joint International Conference on Vector and Parallel Processing*, Sept., 1994.

[4] E. P. Markatos and T. J. LeBlanc. Load Balancing versus Locality Management in Shared Memory Multiprocessors. In *Proceedings of International Conference on Parallel Processing*, August 1992.

[5] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. In *IEEE Transactions on Parallel and Distributed Processing*, pp 379-400, April 94, v 5 n 4.

[6] V. Maslov and W. Pugh. *Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise*. Technical Report CS-TR-3109.1,University of Maryland Institute for Advanced Computer Studies, 1994.

[7] P. Tang and P.-C. Yew. Processor Self-Scheduling for Multiple Nested Parallel Loops. In *Proceedings of International Conference on Parallel Processing*, August 1986.

[8] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.

[9] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SoCs. In *IEEE Design & Test of Computers*, September-October, 2001.