Code Scheduling for Optimizing Parallelism and Data Locality*

Taylan Yemliha[1], Mahmut Kandemir[3], Ozcan Ozturk[2], Emre Kultursay[3], and Sai Prashanth Muralidhara[3]

[1] Syracuse University
[2] Bilkent University
[3] Pennsylvania State University

Abstract. As chip multiprocessors proliferate, programming support for these devices is likely to receive a lot of attention in the near future. Parallelism and data locality are two critical issues in a chip multiprocessor environment. Unfortunately, most of the published work in the literature focuses only on one of these problems, and this can prevent one from achieving the best possible performance. The main goal of this paper is to propose and evaluate a compiler-directed code parallelization scheme, which considers both parallelism and data locality at the same time. Our compiler captures the inherent parallelism and data reuse in the application code being analyzed using a novel representation called the *locality-parallelism graph* (LPG). Our partitioning/scheduling algorithm assigns the nodes of this graph to the processors in the architecture and schedules them for execution. We implemented this algorithm and evaluated its effectiveness using a set of benchmark codes. The results collected so far indicate that our approach improves overall execution latency significantly. In this paper, we also introduce an ILP (Integer Linear Programming) based formulation of the problem, and implement the schedule obtained by the ILP solver. The results indicate that our approach gets within 4% of the ILP solution.

1 Introduction

As chip multiprocessors are finding their ways into commercial market in embedded domain, programming support for these devices is becoming increasingly critical. This support includes language, compiler, and debugging related issues and is likely to receive a lot of attention in the near future.

In a chip multiprocessor based execution environment, two issues are critical to address: *parallelism* and *data locality*. The first of these indicates how well an execution exploits available computation resources. Ideally, one wants to use all available processors at each step of computation if doing so improves

* This research is supported in part by NSF grants CNS #0720645, CCF #0811687, OCI #821527, CCF #0702519, CNS #0720749, by a grant from Microsoft Corporation, and by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme.

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 204–216, 2010.

performance.[1] Data locality, on the other hand, captures how well an execution exercises available memory hierarchy. The concept of data locality is particularly important in the context of chip multiprocessors as the gap between latencies of the on-chip and off-chip accesses is huge. Clearly, one wants to satisfy majority of data references from the higher levels of memory, i.e., those components that are close to processor. In order to achieve good performance in a chip multiprocessor based embedded system, an optimizing compiler has to exploit *both* parallelism and locality in a synergistic fashion.

Unfortunately, most of the published work in the literature focuses only on one of these problems, and this can prevent one from achieving the best possible performance. For example, if locality remains unoptimized, one can expect poor performance at runtime even if all available parallelism is extracted from the application. Similarly, a locality-optimized program that is not parallelized appropriately can result in poor runtime behavior.

The main goal of this paper is to propose and evaluate a compiler-directed code partitioning/scheduling scheme, which considers both parallelism and data locality at the same time. Our target application domain is data-intensive codes that use arrays as the primary data structures. These arrays have affine subscript expressions and are operated using loops with affine bounds. Our compiler captures the inherent parallelism and data reuse in the application code being analyzed using a novel representation called the *locality-parallelism graph*, or LPG for short. It then executes a partitioning/scheduling algorithm on this graph, which assigns the nodes of this graph to the processors in the parallel architecture (a chip multiprocessor). We implemented this algorithm and evaluated its effectiveness using a set of four benchmark codes. An important characteristic of this algorithm is that it has a global view of the computations in the application code. That is, during scheduling, it considers all loop nests and data access patterns before assigning a scheduling slot to a computation. In contrast, most current scheduling efforts are local, i.e., focus on a single loop at a time. The results collected so far indicate that our approach improves execution latency significantly.

In this paper, we also present an ILP (Integer Linear Programming) based formulation of the combined parallelization and data-locality optimization problem. We implemented this ILP solver based solution and compared the results it generated to those obtained using our heuristic approach. The collected experimental results indicate that our approach gets within 4% of the ILP solution.

The chip multiprocessor (CMP) architecture considered in this work is a shared memory based one. In this architecture, multiple CPUs share an on-chip cache space. We also assume the existence of a large off-chip memory space, shared by all processors in the system. The important point to note here is that optimizing for both parallelism and locality is very important in this CMP architecture. In particular, in order to attain good performance, one has to use all available CPUs to the maximum extent allowed by intrinsic data dependencies in the code, and the reused data elements should be caught in the on-chip cache

[1] In some cases, increasing the number of processors used beyond a certain point can degrade performance due to increased inter-processor communication.

space as much as possible (instead of going off chip). This paper demonstrates how data scheduling can be used for this purpose.

The rest of this paper is structured as follows. Section 2 describes our compiler representation, LPG, and Section 3 presents the details of our partitioning/scheduling algorithm. The ILP formulation of the problem is discussed in Section 4. An experimental evaluation of the proposed approach is given in Section 5. Related work is discussed and compared to our work in Section 6, and the paper is concluded in Section 7 with a summary.

2 Locality-Parallelism Graph

Our scheduling algorithm, which targets loop based applications, operates with a *locality-parallelism graph* (*LPG*) of code blocks. This graph captures the dependencies among code blocks and locality among the blocks. An LPG is an acyclic graph $G(V, E_{dep}, E_{loc})$, where V is a set of nodes and E_{dep} and E_{loc} are sets of edges. Each v_i in V represents a *code block* (which will be explained in detail shortly). A directed edge $e_{i,j}$ in E_{dep} from v_i in V to v_j in V means there is a dependency between v_i and v_j (i.e., data produced by v_i is used by v_j). In this case, v_i is an immediate predecessor of v_j, and v_j is an immediate successor of v_i. We denote the set of immediate predecessors of a node v as $Pred_v$, and the set of immediate successors of v as $Succ_v$. A directed edge $e'_{i,j}$ in E_{loc} from v_i to v_j means there is a data reuse, i.e., v_i and v_j share some data between them. The weight $W_{e_{i,j}}$ of edge $e'_{i,j}$ captures the amount of data shared by the two nodes (code blocks). To make our problem formulation simpler, all non-existing edges in an LPG are assumed to have a weight of 0. The set of nodes that share a locality edge with a node v is denoted as Loc_v.

When we have a loop with a large number of iterations, we can rewrite the same loop as a set of loops of fewer iterations, which have the same loop body as the original. For example, if the original loop has n iterations, we can break it into k blocks, each block having roughly n/k iterations. In this context, we call each one of these smaller loops a *code block*. Notice that a given loop (i.e., the set of iterations in it) can be divided into multiple code blocks (unit of scheduling in our approach) and each code block contains a subset of the iterations in that loop. These code blocks can then be executed in parallel, based on the data dependency constraints. The number and size (i.e., the number of iterations) of the code blocks can be arranged to achieve the desired level of granularity. In our case, this code block generation and process of extracting data dependencies among code blocks is carried out by the compiler.

As an example, in the graph in Figure 1, we have five separate loop nests in our code (shown on the left.) The solid lines represent the data dependencies, whereas the dotted lines capture the data reuse edges.[2] We partition each of these

[2] Note that, while a data dependency edge between two nodes means that there is also a data reuse edge between them, the opposite may not always be true. This is because if two code blocks only read the same data, this does not introduce a data dependency but we still have a data reuse between them.

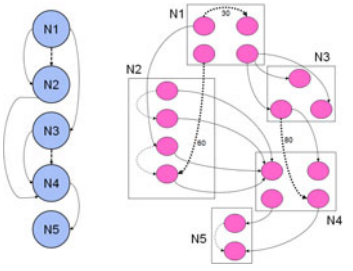

Fig. 1. An example illustrating the concept of LPG

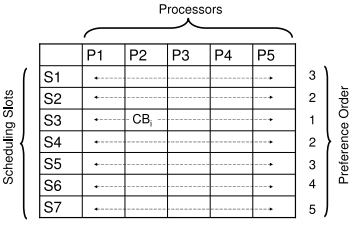

Fig. 2. An example scheduling matrix

loops into smaller loops of reasonable size (i.e., into code blocks). On the right part of this figure, we see the code blocks that are generated by the partitioning of loop nests. Note that the number of edges has increased, since it now shows the dependencies and data reuse between small code blocks, instead of larger nests. This graph on the right (which is our LPG in this case) shows dependencies and data localities at a finer granularity, and is the main compiler based data structure on which our partitioning/scheduling scheme operates. When there is no confusion, in the remainder of this paper, when we mention "block", we mean "code block".

3 Our Approach

As mentioned earlier, our target domain is data intensive computations that operate primarily on array data using nested loops. We assume that the loop bounds and array subscript expressions are affine functions of enclosing loops and loop-independent variables/constants.

We can think of a *schedule* as a two dimensional matrix, where the rows represent scheduling steps (also referred to as the execution steps in this work) and the columns correspond to available CPUs. At the end of scheduling (which is explained below), we fill the entries of this matrix such that both parallelism and data locality are improved.

Our goal is to schedule, considering the CPUs we have in our given CMP, code blocks that have data reuse between them as close together as possible (in time), while respecting data dependencies.

Consider for example the scheduling matrix shown in Figure 2. From the parallelism perspective, we want to fill all the entries in a row (scheduling slots) with code blocks. However, as mentioned earlier, data dependencies among code blocks may not allow such full utilization of scheduling slots. From the data locality perspective on the other hand, we want the code blocks that share data among them to be scheduled in close by scheduling slots. For example, Figure 2 shows the preferable scheduling slots (1 being the most preferable and 5 being the least) for a code block CB_j that has data reuse with code block CB_i if the latter has already been scheduled as shown in the figure.

To accomplish this goal, we designed a heuristic algorithm for resource-constrained scheduling. In this section, the key parts of our algorithm are given as a pseudo-code along with short explanations of the functions implemented. In our approach (which is fully automated), certain data structures are used throughout our algorithm, and are considered global. These are given in Table 1.

Table 1. Global variables

Variable	Definition
$Step_v$	Execution step that node v is scheduled. The value is 0 for unscheduled nodes.
Sch_i	the set of nodes scheduled in execution step i.
$sReady$	the set of nodes that are ready to be scheduled.
$sManda$	the set of nodes that are ready and have to be scheduled as soon as possible.
$sRemain$	the remaining set of nodes that are not ready.
cap	the remaining capacity in the current step.

Before discussing the technical details of our scheduling strategy, let us informally state what it does. Our approach consists of arranging nodes of a given LPG into an execution schedule depending on the amount of data reuse they exhibit and parallelism they have. At each step, a node is scheduled for the current execution step (scheduling slot); the node is selected such that it has the largest amount of data reuse with the nodes already selected for that step, as well as nodes scheduled for previous steps in a weighted fashion (i.e., data reuse within the same step has more weight than that of previous steps). While choosing the first node in a step, its amount of data reuse with ready-to-be-scheduled but unscheduled nodes is also taken into account.

In more technical terms, our approach starts by computing the ASAP (*as soon as possible*) and ALAP (*as late as possible*) values for the nodes of the LPG at hand. An ASAP value for a node v gives the earliest step of execution that v can be scheduled. The ASAP algorithm assigns an ASAP label S_v to each node v. Similarly, in ALAP scheduling, each block is scheduled to start at the latest possible step. An ALAP value for a node v represents the latest step of execution that v can be scheduled. The ALAP algorithm assigns an ALAP label L_v (step index) to each node v. T represents an upper bound on the number of steps. We omit the pseudo codes for the $ASAP()$ and $ALAP()$ procedures since they are well known in literature [19]. Each step of execution has a capacity of p, that is, the number of processors in the system. In other words, at most p code blocks can be scheduled in an execution step.

The procedure $calcSchedule(p, \alpha, \beta, flag)$ in Algorithm 1 calculates the schedule for a system with p processors. It first initializes several variables, populates $sRemain$, and then assigns $ASAP/ALAP$ labels to each node. $insertReadyNodes(1)$ adds the possible starting nodes (nodes that are not dependent on any other nodes) to $sReady$.

Algorithm 1. *calcSchedule*$(p, \alpha, \beta, flag)$

```
procedure CALCSCHEDULE(p, α, β, flag)
    for all v ∈ V do
        Step_v ← 0
        sRemain ← sRemain ∪ {v}
    end for
    cs ← 0                                              ▷ current step
    ASAP()
    ALAP()                                              ▷ Assign ASAP/ALAP values to each node
    sReady ← ∅                                          ▷ initialize ready nodes set
    insertReadyNodes(1)
    while (sReady ≠ ∅) do
        cs ← cs + 1                                     ▷ next execution step
        Sch_cs ← ∅
        cap ← p                                         ▷ remaining capacity in current step
        doMandatory(cs)
        while (cap > 0) and (sReady ≠ ∅) do
            myMax ← −1
            myNode ← null
            for all v ∈ sReady do
                ss ← 0                                  ▷ sReady score
                if (flag and (Sch_cs = ∅)) then
                    ss ← calcScore_sReady(v, p − 1)
                end if
                ts ← calcScore_Sch(v, α, cs) + β * ss
                if ts > myMax then                      ▷ total score
                    myMax ← ts
                    myNode ← v
                end if
            end for
            scheduleNode(myNode, cs)
        end while
        insertReadyNodes(cs + 1)
    end while
end procedure
```

The rest of the code is the main *while* loop, which iterates as long as there are ready nodes (code blocks) to be scheduled. At each iteration, it first goes on to next execution step, initializes variables, and calls $doMandatory$ (schedules nodes that would otherwise increase the total number of execution steps.) At each iteration of the inner *while* loop, the ready node with the highest score is selected and scheduled, as long as there are ready nodes and there is room in the current execution step to accommodate new code block(s). When either condition is $false$, $insertReadyNodes(cs + 1)$ loads *sReady* with nodes ready for the next execution step and goes back to the next iteration of the main *while* loop.

The selection of the nodes from *sReady* is performed according to the score calculated by *calcScore_Sch*. The boolean $flag$ enables contribution to the score by *calcScore_sReady*, in the case of current step being empty. If the flag is down, the first node of the step is chosen, solely based on data reuses it exhibits with nodes scheduled in previous steps. Otherwise, a constant (β) times the *calcScore_sReady* score (a node's locality with $p - 1$ other nodes in *sReady*) is added to the total score. Note that the use of the parameters $flag$, α and β enables us to generate multiple heuristics and fine-tune our algorithm.

The procedure $insertReadyNodes(s)$ in Algorithm 2 scans the set of nodes *sRemain* and determines if any of the nodes are ready for step s (i.e., all its

Algorithm 2. *insertReadyNodes*(s)

```
procedure INSERTREADYNODES(s)
    for all node v in sRemain do
        if ((S_v ≤ s) and (L_v ≥ s) and (∀v_i ∈ Pred_v. Step_{v_i} > 0)) then
            sReady ← sReady ∪ {v}
            sRemain ← sRemain − {v}
        end if
    end for
end procedure
```

predecessors have already been scheduled, and step s lies between its ASAP and ALAP labels). It then puts all ready nodes in the set of ready nodes, $sReady$, and deletes them from $sRemain$.

The procedure $doMandatory(s)$ in iterates through the nodes in $sReady$ to find the nodes that have an $ALAP$ label L_v such that $L_v \leq s$ (the given step) and adds them to $sManda$. It then schedules the nodes in $sManda$ in the order of non-decreasing $ALAP$ values as long as there are nodes in $sManda$ and there is available capacity in step s.

The procedure $scheduleNode(v, s)$ schedules node v in execution step s. It updates Sch_s and $Step_v$ accordingly, deletes the node from $sReady$, and decreases the remaining capacity of the execution step.

The function $calcScore\_sReady(v_i, c)$ in Algorithm 3 is optionally used when it is time to pick the first node for a step, since the amount of data reuse (i.e., the number of data elements shared) with other nodes already scheduled on the same step is not sufficiently high. Instead, it calculates a total weight value for v_i, based on its amount of data reuse with other unscheduled ready nodes. The parameter c is used as an upper bound on the number of nodes considered, since there can be at most p nodes scheduled in a step. Basically, the function returns the sum of highest t locality weights that v_i shares with other ready nodes. The actual number of nodes is given by $t = min(c, \mid Loc_{v_i} \cap sReady \mid)$.

The weight value returned by $calcScore\_sReady(v_i, c)$ is given by the formula:

$$\sum_{v_j \in (Loc_{v_i} \cap sReady)} W_{e_{i,j}},$$

where v_j is one of the t nodes in $sReady$ with the highest data reuse with respect to v_i.

The function $calcScore\_Sch(v_i, \alpha, s)$ calculates the amount of weighted data reuse between v_i and already scheduled nodes both in the same step (s) and previous steps. The weight value returned by $calcScore\_Sch(v_i, \alpha, s)$ is given by the formula:

$$\sum_{v_j \in Loc_{v_i}} [sgn(Step_{v_j}) * W_{e_{i,j}} / (\alpha * (s - Step_{v_j}) + 1)]$$

The signum function in the formula prevents contribution from unscheduled nodes, and is defined as follows:

Algorithm 3. *calcScore_sReady*(v_i, c)

```
function CALCSCORE_SREADY(v_i, c)
    k ← 0
    for all (v_j ∈ Loc_{v_i}) do
        if v_j ∈ sReady then
            k ← k + 1
            sReadyLocs[k] ← W_{e_{i,j}}
        end if
    end for
    sort sReadyLocs
    wt ← 0
    for l ← 1, Min(c, k) do
        wt ← wt + sReadyLocs[k − l + 1]
    end for
    return wt
end function
```

$$sgn(x) = \begin{cases} 0 : x = 0 \\ 1 : x > 0 \end{cases}$$

Different α values change the effect of step difference (e.g., α=0 ignores execution steps and uses the locality value directly, whereas a large α value concentrates on the current execution step).

4 ILP Formulation of the Problem

In order to see how close our heuristic comes to the optimal, we also implemented an ILP (Integer Linear Programming) based solution to the problem, and performed experiments with it. This section gives the details of our ILP based solution. Table 2 lists the constant terms used in our ILP formulation. We used lp_solve [2], a free ILP tool, to formulate and solve our 0-1 ILP problem. Although ILP generates an optimal result (under the assumptions made), the time complexity of ILP prohibits practical usage in most cases. The computation of the solutions for the ILP problems mentioned below took days on average and more than a week in one case. Therefore, it is very important to explore heuristic solutions for this combined parallelism-data locality problem.

A 0-1 ILP problem is a special kind of ILP problem, where each variable can only take the value of 0 or 1. In our case, we have only one type of 0-1 solution variable, $X_{i,l}$, which indicate whether v_i is scheduled on step l. To make our presentation clear, we use the expression $Step_{v_i}$ to represent the execution step

Table 2. Constants used in our ILP formulation

Constant	Definition
p	number of processors
$W_{e_{i,j}}$	The weight of edge $e_{i,j}$
S_{v_i}	ASAP value of operation v_i
L_{v_i}	ALAP value of operation v_i

that the code block v_i is scheduled. $Step_{v_i}$ is expressed in terms of the $X_{i,l}$ variables and ILP constants as follows:

$$Step_{v_i} = \sum_{l=S_{v_i}}^{L_{v_i}} (l * X_{i,l}).$$

Our objective is to find the execution step that each node is to be scheduled, such that, nodes that exhibit high data reuse with each other are scheduled as close to each other as possible. In our formulation, the 0-1 variables $X_{i,l}$ are used to capture this information. In other words, we want to minimize the scheduling step distance between nodes with data locality. Therefore, we can express our objective function as follows:

$$\text{Minimize} \sum_{e_{i,j}} (|Step_{v_i} - Step_{v_j}| * W_{e_{i,j}}),$$

where $e_{i,j} \in E_{loc}$ (e.g. $e_{i,j}$ is a locality edge).

We have three types of constraints in our problem.

1. The execution step $Step_{v_i}$ for code block v_i is unique for all $i \in \{1, ..., n\}$. As a result, for a given i, only one of the $X_{i,l}$ variables will take a value of 1, and the rest will be 0, which can be formulated as follows:

$$\sum_{l=S_{v_i}}^{L_{v_i}} (X_{i,l}) = 1.$$

   For example, if $S_{v_1} = 3$ and $L_{v_1} = 5$, then $X_{1,3} + X_{1,4} + X_{1,5} = 1$ is a constraint in our ILP formulation.
2. Sequencing relations must be satisfied. If a code block v_j depends on v_i, then v_j should be scheduled at a later step than v_i. This constraint can be expressed as follows: $Step_{v_j} > Step_{v_i}$, where $e_{i,j} \in E_{dep}$ (e.g., $e_{i,j}$ is a dependency edge).
3. Since we have p processors, at most p code blocks can be scheduled at an execution step. In other words, for any given step l, the sum of $X_{i,l}$ values cannot exceed p. Note that the actual number of nodes scheduled at a step can be less than p due to data dependencies between code blocks. Therefore, for each step l, we include the following constraint in our formulation:

$$[\sum_{i \in \{j | S_{v_j} \leq l \leq L_{v_j}\}} X_{i,l}] \leq p.$$

The above mentioned constraints and objective function constitute our ILP formulation of the problem. In this formulation, the nodes that do not exhibit data reuse are not part of the objective function, and are therefore scheduled based solely on data dependencies, or arbitrarily if they have none.

5 Experiments

We implemented the algorithm explained in this paper as a software tool which takes an LPG, as well as α, β and the number of processors in the CMP as input

parameters. The tool parses the graph and applies our heuristic algorithm with the given parameters to obtain the data locality-optimized parallel execution schedule.

Table 3. Benchmark codes used in our experimental evaluation

Benchmark Name	Source	Number of Nodes
adi	Livermore	40
bmcm	Perfect Club	23
tsf	Perfect Club	26
vpenta	Spec92	28

We used four data-intensive, array-dominated benchmarks to test our algorithm, and performed experiments on three hardware platforms (with 2, 4 and 8 processors.) Table 3 lists the benchmark codes we used and the number of nodes in their LPGs. For our tests, we simulated the hardware and OS using Simics [4]. Simics is a simulation toolset for multi-processor systems and allows building a binary-compatible instance of the target hardware, which operates completely within a virtualized environment running on standard PCs.

For each of the hardware platforms, the following operations were performed. We applied our algorithm to the LPG of each benchmark. The above mentioned ILP formulation of each of these problems produced an alternate schedule. We used the Intel C++ Compiler 10 [1] and OpenMP API [3] to compile the codes resulting from these schedules (for both heuristic scheme and ILP solver based scheme). We also compiled the original code both without parallelization and using the Intel compiler's own parallelization mechanism. For our tests, the default values of $\alpha = \beta = 1$ were used. We also made experiments with other values of the α and β parameters, but the results were very close to those shown here for the $\alpha = \beta = 1$ case. More specifically, the difference between the result obtained with different α, β values were within 4%.

On each platform, we obtained four results for each benchmark. Figures 3, 4 and 5 show the results of the experiments with 2, 4 and 8 processors, respectively. The bars show the speeds *normalized* with respect to the result given by our heuristic algorithm. The normalized value for each version is computed as the

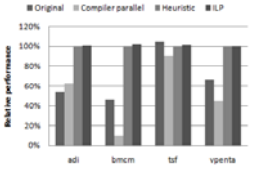

Fig. 3. Results for the 2 processor case

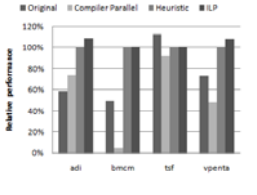

Fig. 4. Results for the 4 processor case

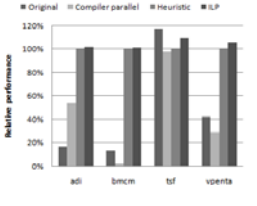

Fig. 5. Results for the 8 processor case

ratio of the heuristic result's execution time to that version's execution time. The four bars for each benchmark in the figures represent the performances achieved by the following scheduling schemes:

1. **Original**: Original program with no scheduling. This represents the sequential code without exploiting loop level parallelism.
2. **Compiler Parallel**: The original code compiled by using the compiler's [1] own parallelization mechanism. While we obviously do not know the details of this parallelization strategy, it is reasonable to assume that it represents state-of-the-art in industry.
3. **Heuristic Parallel**: The code scheduled by our heuristic scheduling approach (Section 4).
4. **ILP**: The code scheduled based on the solution returned by the ILP based formulation (Section 5).

Our first observation is that, our algorithm performs better than the original in all but one of the benchmarks. This exception is due to that benchmark having a small body of code, which makes the synchronization overhead brought by parallelization significant. We also note that our scheduling algorithm performs better than the compiler's own parallelization mechanism in all cases, and is very close to the result achieved by the ILP solver. The overall speedups achieved by our algorithm with respect to the original code are 1.62, 1.50 and 4.21 for the 2, 4 and 8 processor cases, respectively. By comparison, the overall speedups achieved by the ILP based solution with respect to the original code are 1.63, 1.54 and 4.29 for the 2, 4 and 8 processor cases, respectively. As stated earlier however, ILP based scheduling may not be a viable option in some cases due to enormous solution times it requires.

6 Related Work

In this section, we first evaluate the previous work on parallelization then we revise the efforts on data locality. Parallelism can be obtained at different levels of abstraction. Instruction-level parallelism is exploited by high-performance microprocessors, whereas data-level parallelism is utilized in nested loops using compilers. Similarly, task-level parallelism can be found in many embedded applications. To exploit the data-level parallelism, Kadayif et al [12] proposed to use different number of processor cores for each loop nest to obtain energy savings. This way idle processors are switched to a low-power mode to increase the energy savings. Hogstedt et al [11] predict the execution time of tiled loop nests and use this prediction to automatically determine the tiling parameters that minimizes the execution time. Arenaz et al [5] exploit coarse-grain parallelism by a gated single assignment (GSA) based approach with complex computations. Yu and D'Hollander [23] construct an iteration space dependency graph to visualize a 3D iteration space. Beletskyy et al [6] adopt a hyperplane-based representation to apply on transformation matrices with both uniform and non-uniform dependences. Lim et al [18] employ affine partitioning to maximize parallelism

with minimum communication overhead. Ozturk et al [21] focus on optimizing parallelism in chip multiprocessors using constraint networks.

Two major techniques to exploit locality are loop transformations and data transformations. Wolf and Lam [22] define reuse vectors and reuse spaces. Moreover, they use these concepts to implement an iteration space optimization technique. Similarly, Li [17] uses reuse vectors to detect the dimensions of loop nest that carry reuse. Tiling [9,14,15] is another loop based locality enhancing technique. On the data transformation side, in [20], authors generate the code with a given data transformation matrix. Kandemir et al [13] implement an explicit layout representation, whereas [16] focuses more on memory consumption reduction due to a layout transformation. There are also efforts to combine data and loop transformations. Among these is one of the first papers [8] that offers a scheme which unifies loop and data transformations.

As compared to these prior studies, we target chip multiprocessors where processors share an on-chip cache and propose a scheduling scheme for improving both data reuse and parallelism in a synergistic manner.

7 Conclusion

Increasing use of chip multiprocessors in embedded computing domain makes automated software support a primary concern for programmers. In particular, compiler plays an important role since it shapes the code behavior as well as data access pattern. Targeting chip multiprocessors and loop-intensive computations, we propose a novel compiler-based loop scheduling scheme with the goal of exploiting both parallelism and locality. This paper describes our strategy and evaluates it using a set of four application codes. The scheduled generated by our algorithm are compared to those obtained by an ILP based scheduler and the original codes. The experimental results we collected are promising and indicate that our approach achieves better results than the commercial compiler and the improvements we obtain are close to those obtained using the ILP solver based scheduler. Our ongoing work on loop scheduling includes porting our strategy to CMPs with private data caches as well as scratch-pad memories.

References

1. Intel C++ compiler 10.0 for Linux, `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/277618.htm`
2. lp_solve, `http://sourceforge.net/projects/lpsolve`
3. OPENMP: Simple, portable, scalable SMP programming, `http://www.openmp.org`
4. Simics, `http://www.simics.com`
5. Arenaz, M., Tourino, J., Doallo, R.: A gsa-based compiler infrastructure to extract parallelism from complex loops. In: Proceedings of ICS (2003)
6. Beletskyy, V., Drazkowski, R., Liersz, M.: An approach to parallelizing non-uniform loops with the Omega Calculator. In: Proceedings of ICPCEE (2002)

7. Chen, G., Ozturk, O., Kandemir, M., Kolcu, I.: Integrating loop and data optimizations for locality within a constraint network based framework. In: Proceedings of ICCAD (2005)
8. Cierniak, M., Li, W.: Unifying data and control transformations for distributed shared-memory machines. In: Proceedings of PLDI (1995)
9. Coleman, S., McKinley, K.S.: Tile size selection using cache organization and data layout. In: Proceedings of PLDI (1995)
10. Dick, R.P., Rhodes, D.L., Wolf, W.: Tgff: task graphs for free. In: Proceedings of CODES (1998)
11. Hogstedt, K., Carter, L., Ferrante, J.: On the parallel execution time of tiled loops. In: IEEE TPDS, vol. 14(3) (2003)
12. Kadayif, I., Kandemir, M., Karakoy, M.: An energy saving strategy based on adaptive loop parallelization. In: Proceedings of DAC (2002)
13. Kandemir, M., Ramanujam, J., Choudhary, A.: A compiler algorithm for optimizing locality in loop nests. In: Proceedings of ICS (1997)
14. Kodukula, I., Ahmed, N., Pingali, K.: Data-centric multi-level blocking. In: Proceedings of PLDI (1997)
15. Lam, M.D., Rothberg, E.E., Wolf, M.E.: The cache performance and optimizations of blocked algorithms. In: Proceedings of ASPLOS (1991)
16. Leung, S., Zahorjan, J.: Optimizing data locality by array restructuring. Technical Report TR-95-09-01 (1995)
17. Li, W.: Compiling for NUMA parallel machines. PhD thesis, Cornell University, Ithaca, NY, USA (1993)
18. Lim, A.W., Cheong, G.I., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: Proceedings of ICS (1999)
19. DeMicheli, G.: Synthesis and optimization of digital circuits. McGraw Hill, New York (1994)
20. O'Boyle, M.F.P., Knijnenburg, P.M.W.: Nonsingular data transformations: Definition, validity, and applications. Int. J. Parallel Program. 27(3) (1999)
21. Ozturk, O., Chen, G., Kandemir, M.: A constraint network based solution to code parallelization. In: Proceedings of DAC (July 2006)
22. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Proceedings of PLDI (1991)
23. Yu, Y., D'Hollander, E.H.: Loop parallelization using the 3d iteration space visualizer. Journal of Visual Languages and Computing 12(2) (2001)