Locality-Conscious Process Scheduling in Embedded Systems*

I. Kadayif, M. Kandemir
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802, USA
{kadayif,kandemir}@cse.psu.edu

I. Kolcu
Computation Department
UMIST
Manchester M60 1QD, UK
i.kolcu@co.umist.ac.uk

G. Chen
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802, USA
gchen@cse.psu.edu

ABSTRACT

In many embedded systems, existence of a data cache might influence the effectiveness of process scheduling policy significantly. Consequently, a scheduling policy that takes inter-process data reuse into account might result in large performance benefits. In this paper, we focus on array-intensive embedded applications and present a locality-conscious scheduling strategy where we first evaluate the potential data reuse between processes, and then, using the results of this evaluation, select an order for process executions. We also show how process codes can be transformed by an optimizing compiler for increasing inter-process data reuse, thereby making locality-conscious scheduling more effective. Our experimental results obtained using two large, multi-process application codes indicate significant runtime benefits.

1. INTRODUCTION

The success of any embedded system depends largely on the degree and effectiveness of the coordination between hardware and software. In fact, many codesign strategies (e.g., [3, 4, 2]) try to come up with hardware and software architectures that work well together. In this context, operating system (OS) plays an important role in closing the gap between hardware and software and interfacing them. It achieves this through several inter-related activities such as resource managing, scheduling and context switching, and protection. Process scheduler, in particular, is an important part of an OS as it is the major component that decides how processor(s) will be shared among processes. The process scheduling in embedded systems [12] is different from that in general purpose computing in at least two ways. First, the scheduler in an embedded system can be fully customized based on the applications at hand. Second, since most of embedded systems execute either a single or a small set of applications, the process granularities can be smaller than those in their general-purpose counterparts. It should also be stressed that in many cases, due to code clarity and maintenance purposes, it might be more beneficial to program a large application as multiple inter-related processes. Consequently, it is not uncommon to divide a large embedded application into multiple processes and schedule these processes using a customized scheduler.

Data caches are being increasingly employed in embedded designs. Previous work [6] shows that compiler based optimizations can be very important in increasing the effectiveness of cache memories. In an OS-based environment, when context switches occur frequently (either due to fine granular processes or due to frequent interrupts), flushing the contents of cache memory during context switches may not be a wise option. This is particularly true if the processes in the system share data structures; e.g., they are extracted from the same application. Instead, a better alternative would be making the process scheduler cache-conscious and exploiting the contents of the data cache as much as possible when moving from one process to another during execution.

In this paper, we focus on array-intensive embedded applications and present a technique to schedule processes in a locality-conscious (cache-sensitive) manner. In this approach, we view a given application as a set of processes that communicate with each other using shared memory.[1] Since these processes share data, the order in which they are activated may influence the data cache behavior significantly. Our two-step optimization strategy first, using an optimizing compiler and a polyhedral library, evaluates the potential data reuse between processes, and then (using the results of this evaluation) selects an order for process executions (i.e., a schedule). While our approach is in essence a static scheduling technique, the experimental results obtained using two large, multi-process application codes indicate significant runtime benefits. We also show in this paper how process codes can be transformed (e.g., using a compiler) for increasing inter-process data reuse, thereby making locality-conscious scheduling more effective. In other words, the work described in this paper also demonstrates how an optimizing compiler can be used in increasing OS performance.

The rest of this paper is organized as follows. In Section 2, we give an overview of our approach, and present our assumptions. In Section 3, we explain a strategy for evaluating data reuse between processes. In Section 4, we present our locality-conscious process scheduling strategy. In Section 5, we present our experimental setup and report performance data. Finally, in Section 6, we conclude the paper with a summary of our major contributions and a brief outline of future research.

2. OVERVIEW OF OUR APPROACH

Many embedded systems execute a number of processes concurrently. A problem occurs in a cache-based environment when data cache locality created by one process during its time quantum is not preserved until its next time quantum. This may occur frequently, for example, when an intervening process execution causes eviction of data cache contents belonging to a previous process.

*This work is supported in part by NSF Career Award #0093082.

CODES'02, May 6-8, 2002, Estes Park, Colorado, USA.

[1] As a matter of fact, in most OS literature, these light-weight processes are called threads.

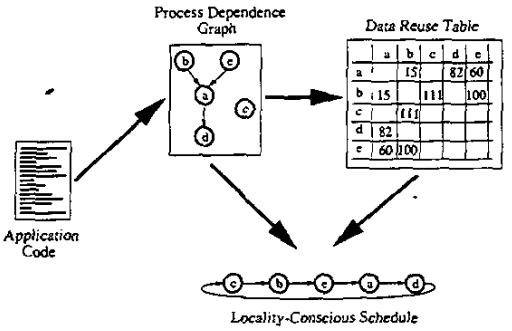

Figure 1: Overview of our optimization strategy.

While a similar problem occurs in instruction caches as well, there, a careful mapping of processes to memory can help reduce the severity of this problem significantly (by minimizing the number of instruction cache conflicts between different processes). In the instruction cache case, this is a viable option as the sizes of processes (measured in bytes or words) are not very large in general, and in most cases it might even be possible to come up with a conflict-free process-to-memory mapping. We refer the reader to [14] and [8] for elegant process mapping strategies that target instruction caches. Li and Wolfe [9] present a model for estimating the performance of multiple processes sharing a cache.

Unfortunately, as compared to process sizes, sizes of data manipulated by processes might be very large. This is particularly true for embedded data-intensive image and video processing applications where small nested loops can manipulate large quantities of data. Given that the most data caches used in embedded systems are relatively small in size, in most cases, it may not be possible to find a suitable data space-to-memory mapping that maintains data locality of a given process between its successive quanta.

In this paper, we propose an alternative solution for maintaining data locality of processes in a data cache based execution environment where processes are extracted from the same application. Instead of trying to map data spaces of processes in memory such that the cache interference will be minimized, our approach tries to reuse the data in the cache as much as possible. Our objective is to modify process scheduler such that the successively scheduled processes exploit as much data cache locality as possible. An important question then is to determine the potential data reuse between two processes. In order to do this, we employ a compiler-oriented data reuse analysis. In other words, using an optimizing compiler, we analyze source codes of processes, and come up with a suitable order of execution that maximizes data cache locality. Our strategy is summarized in Figure 1. The application code is first analyzed and, based on this analysis, a process dependence graph is built. Then, an optimizing compiler evaluates the amount of data reuse between processes and constructs a data reuse table (i.e., a table that holds the amount of data reuse between processes). Finally, using the process dependence graph and data reuse table, we determine a legal, layout-conscious schedule for processes. We also show that in some cases applying loop transformations to process codes may improve inter-process locality, thereby enhancing the effectiveness of locality-conscious scheduling.

We assume (mainly for simplicity) that each process consists of a loop nest and that the data dependences between processes are represented using inter-process dependence graph (see Figure 1), PDG. In this graph, each node represents a process (loop nest) and a directed edge from one node to another indicates that the process represented by the second node is dependent on the process represented by the first node (i.e., it cannot start execution until the first one finishes). Note that any schedule constraint other than data dependences are also captured in this graph. We further assume that all processes run on a single CPU and each process is run to completion. Finally, it is assumed that when all the processes in the PDG are executed, the execution jumps to the first process and the entire execution sequence is repeated.

Our focus in this paper is on affine programs [5] as defined by Feautrier. Data structures in these programs are restricted to be arrays and scalar variables, and control structures are limited to sequencing and nested loops. Each iteration of a given nested loop is represented by an iteration vector, $\vec{I}$, which contains the values of the loop indices from outermost position to innermost. Each array reference to an m-dimensional array in a nested loop that contains k loops (i.e., a k-level nest) is represented by $f(\vec{I}) = L\vec{I} + \vec{o}$, where $\vec{I}$ is the iteration vector. For a specific $\vec{I} = \vec{J}$, the data (array) element $L\vec{J} + \vec{o}$ is accessed. In this representation, the $m \times k$ matrix L is called the access (or reference) matrix and the m-dimensional vector $\vec{o}$ is called the offset (or constant) vector [11, 13]. All values that can be assumed by an iteration vector $\vec{I}$ define an iteration space, $\mathcal{I}$.

The application of a loop transformation represented by a square, non-singular matrix T can be accomplished in two steps [13]: (i) rewriting loop body and (ii) rewriting loop bounds. The first step is quite easy. Assuming that $\vec{I}$ is the vector that contains the original loop indices and $\vec{I'} = T\vec{I}$ is the vector that contains the new (transformed) loop indices, each occurrence of $\vec{I}$ in the loop body is replaced by $T^{-1}\vec{I'}$ (note that T is invertible). In other words, each reference represented by $L\vec{I}+\vec{o}$ is transformed to $LT^{-1}\vec{I'}+\vec{o}$. Determination of the new loop bounds is more complicated and in general may require the use of Fourier–Motzkin elimination [13].

3. INTER-PROCESS LOCALITY EVALUATION

The objective of our inter-process locality evaluation strategy is to find out the maximum potential locality between two processes. We start by checking whether the two processes have any common array. If not, then there is no data reuse between these processes, and (from the data locality perspective) there is no benefit in scheduling these two processes one after another. On the other hand, if there exists at least one common array between the processes, we proceed as explained in the following subsections.

We consider two different strategies for addressing this inter-process reuse evaluation problem. In the first strategy, we assume that no transformation will be performed on the source codes of the processes. That is, the source codes will be used as they are without any modification. In the second strategy, we assume that we have the flexibility of modifying the source codes (maintaining all intrinsic data dependences) if doing so leads to better exploitation of inter-process locality.

3.1 Evaluating Locality Without Transformations

The problem can be defined as one of determining the amount of data reuse between two processes that can be scheduled one after another. Since we assume that, in each activation, a process executes its code to completion (unless an external interrupt occurs), one simple measure of inter-process data reuse is the number of array elements shared between processes. Let us first focus on a simple case where each process code contains a single array (U) and a single reference to it. Assume that we have two processes, a and b, and that $U[f(\vec{I})]$ occur in the first process and $U[g(\vec{J})]$ occur in the second one, where $\mathcal{I}$ and $\mathcal{J}$ are the corresponding iteration spaces and, $\vec{I} \in \mathcal{I}$ and $\vec{J} \in \mathcal{J}$ are the iteration vectors. In order to express the number of elements shared between these two references, we use Presburger Formulas. Presburger formulas are set expressions that contain affine constraints, the usual logical connectives (i.e., 'and', 'or', and 'not' connectives), and $\forall$ (universal) and $\exists$ (existential) quanti-

fiers. Note that many concepts associated with nested loops and array accesses can be expressed using Presburger formulas [7]. For example, the set of iterations executed by a loop nest can be described as $\{[i_1, ..., i_m] : L_1 \leq i_1 \leq M_1$ and ... and $L_m \leq i_m \leq M_m\}$, where L_j and M_j are the affine lower and upper bounds, respectively, for loop index i_j. In our context, we can express the set of elements shared between the two processes mentioned above (a and b) as:

$$S_{a,b}(U) = \{\vec{s} \mid \exists \vec{I} \in \mathcal{I} \text{ and } \exists \vec{J} \in \mathcal{J} \text{ such that } \vec{s} = f(\vec{I}) = g(\vec{J})\}.$$

So, the *degree of reuse* (DR) between these two processes (due to array U) can be expressed as:

$$|S_{a,b}(U)|,$$

that is, the number of elements in $S_{a,b}(U)$. While in general computing the number of elements in $S_{a,b}(U)$ is expensive, there are several polyhedral tools from academia and industry that work very well (fast) in practice. One such tool is the Omega library [7]. Using the Omega library we can generate a loop that enumerates the elements that belong to a Presburger set, and by executing this loop we can count the number of elements it contains. Our experience shows that the Omega Library is very fast in practice. Note also that our locality evaluation activity is an off-line strategy; that is, we can afford to spend more cycles (than a runtime strategy) for obtaining a good schedule (with large runtime benefits).

As an example, let us assume that the reference in the first process is $U[i+j][k]$, where $1 \leq i, j, k \leq n$ and the reference in the second process is $U[n-2][k]$, where $1 \leq k \leq n$. Here, i, j, and k are loop variables (indices). Then, the common elements can be expressed as:

$$\begin{aligned} S_{a,b}(U) = \{&[s_1][s_2] \mid (1 \leq i \leq n) \text{ and } (1 \leq j \leq n) \text{ and} \\ &(1 \leq k \leq n) \text{ and } (s_1 = i + j = n - 2) \text{ and } (s_2 = k)\}. \end{aligned}$$

In general, let as assume that one of the processes has x references to array U (denoted $U[f_1(\vec{I})]$, $U[f_2(\vec{I})]$, ..., $U[f_x(\vec{I})]$), and the other has y references (denoted $U[g_1(\vec{J})]$, $U[g_2(\vec{J})]$, ..., $U[g_y(\vec{J})]$). Then, we have:

$$\begin{aligned} S_{a,b}(U) = \{&\vec{s} \mid \exists k, l, \vec{I}, \vec{J} \text{ such that } (1 \leq k \leq x) \text{ and} \\ &(1 \leq l \leq y) \text{ and } (\vec{I} \in \mathcal{I}) \text{ and } (\vec{J} \in \mathcal{J}) \text{ and} \\ &(\vec{s} = f_k(\vec{I}) = g_l(\vec{J}))\}. \end{aligned}$$

Note that in general there might be more than one common array between two processes. Assuming that U_1, U_2, ..., U_r are the common arrays between process a and process b, we can define the overall degree of reuse between these two processes as:

$$|S_{a,b}|,$$

where

$$\begin{aligned} S_{a,b} &= S_{a,b}(U_1) \cup S_{a,b}(U_2) \cup ..... \cup S_{a,b}(U_{r-1}) \cup S_{a,b}(U_r) \\ &= \bigcup_{i=1}^{r} S_{a,b}(U_i) \end{aligned}$$

Note that our approach computes $|S_{a,b}|$ for each process a and process b that can be scheduled (without breaking inter-process dependences) one after another. Note also that since we do not consider any transformation of the codes of the processes, whether we schedule process a followed by process b or process b followed by process a does not make a difference in computing $|S_{a,b}|$. Once the $|S_{a,b}|$ values have been found, we can fill the entries in the data reuse table (Figure 1).

3.2 Evaluating Locality Using Transformations

In some cases, $|S_{a,b}|$ may not be a very accurate measure when the data manipulated by a given process is very large as compared to the cache capacity. For example, consider the following code fragment that belongs to a process (called a):

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c = c + U[i][j];
```

In this fragment, if the cache capacity is much smaller than n^2 (array size), only a small subset of the array (e.g., the last couple of rows) might stay in the cache after the execution. Consequently, if we have another process (called b) with the following nest:

```
for (i=0; i<n/8; i++)
  for (j=0; j<n; j++)
    d = d + (U[i][j]*U[i][j]);
```

we might not be able to take advantage of the reuse between these two processes if we schedule them one after another. This is because this second loop access only the first $n/8$ rows of the array. With a small cache capacity, these elements may not even be in the cache after the execution of the loop in process a. Note, however, that the strategy described in Section 3.1 determines (for this case) a degree of reuse $|S_{a,b}| = n^2/8$, which indicates a significant amount of data reuse between these two processes. So, the strategy in Section 3.1 is not very accurate in capturing locality when the volume of data manipulated is much larger than the available cache capacity.

There are two apparent solutions to this problem. First, instead of scheduling first a and then b, we can schedule first b and a. The problem with this approach is that, in general, data dependences between two processes do not permit such a scheduling. The second solution is to transform the loop of the first process in such a fashion that the transformed loop, after the execution, leaves in the cache the elements that will be accessed by the second process. We can achieve this by transforming the said code to the following fragment:

```
for (i=n-1; i ≥ 0; i--)
  for (j=n-1; j ≥ 0; j--)
    c = c + U[i][j];
```

In this code fragment, the array is traversed in the opposite direction of the original code. Therefore, when the loop finishes its execution, the cache contains (depending on the cache size) the elements that will be needed by process b. If, instead, the process b had the following loop:

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    e = e + (U[i][j]*U[i][j]);
```

a suitable solution would be transforming this loop to the following form (without modifying the original code of process a):

```
for (i=n-1; i ≥ 0; i--)
  for (j=n-1; j ≥ 0; j--)
    e = e + (U[i][j]*U[i][j]);
```

This discussion reveals that in cases where data manipulated by processes cannot be fit in the cache, it might be better to use loop transformations to increase data reuse between the processes. It should be mentioned that in considering two processes a and b, when we take into account the possibility of transforming loops, we may need to compute both $S_{a,b}$ and $S_{b,a}$ separately. In computing $S_{a,b}$, our approach tries to transform process b (considering the footprint of a in the cache after execution), whereas in computing $S_{b,a}$, it tries to transform a (considering the footprint of b in the cache after execution). This is reasonable as $S_{a,b}$ (resp. $S_{b,a}$) is only meaningful when process a (resp. process b) is to be scheduled immediately before process b (resp. process a).

To capture the footprint of a process in the cache after its execution, we use a representation called *footprint vector*. Informally, for a given reference, its footprint vector indicates how the array is traversed by the process using that reference. Let us assume that the

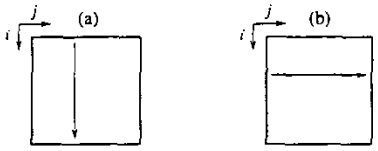

Figure 2: The access patterns implied by (a) $\vec{v}^i = [1 \quad 0]^T$, and (b) $\vec{v}^j = [0 \quad 1]^T$.

reference in question has an access matrix L and an offset vector $\vec{o}$. Assume further that $\vec{I}$ denotes a specific loop iteration and $\vec{I}_{+j}$ denotes the loop iteration that has the same entries as $\vec{I}$ except that the loop index value in the jth position is increased by 1 (we assume that this increase does not exceed the upper bound of this loop). As an example, if $\vec{I} = [4 \quad 6]^T$, then $\vec{I}_{+i} = [5 \quad 6]^T$ and $\vec{I}_{+j} = [4 \quad 7]^T$ (here, i is the outer loop whereas j is the inner).

We can define the footprint vector for this reference in the jth position as:

$$\vec{v}^j = (L\vec{I}_{+j} + \vec{o}) - (L\vec{I} + \vec{o}) = L(\vec{I}_{+j} - \vec{I}) = l_j$$

where l_j is the jth column of L. For example, in the original loop of the process a above, we have:

$$L = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad \vec{o} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Therefore, the footprint vectors are:

$$\vec{v}^i = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{and} \quad \vec{v}^j = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Note that what this $\vec{v}^i$ means is that if we keep the loop index j at a fixed value and keep increasing the value of the loop index i, we traverse the array from the first row to the last. Similarly, the vector $\vec{v}^j$ given above means that if we keep increasing the value of j by keeping i at a specific value, we traverse the array from left to right. These two scenarios are illustrated in Figures 2(a) and (b).

Note that in general there might be multiple references to a given array in the same loop. Consequently, we need to use a *footprint matrix* (for each position) instead of a footprint vector. As an example, if we have two references, $U[i][j+1]$ and $U[j][i+j]$, the footprint matrices are as follows:

$$V^i = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad V^j = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

In cases where we have a footprint matrix for a given loop position instead of a footprint vector, we build a *summary footprint vector* for that position. In mathematical terms, we need to generate a single vector from several vectors. There are at least three alternative strategies for doing that. The first alternative selects the most frequently used reference to the array and uses its footprint vector as the summary footprint vector for the entire array. The second alternative selects the footprint vector of the last reference to the array (in execution) as the summary footprint vector for the entire array. The third alternative combines multiple footprint vectors into a summary footprint vector using some form of vector operator. In this work, we used simple vector combination as the operator. In the following discussion, when we mention footprint vector it can also mean summary footprint vector; the intend should be clear from the context.

It should be emphasized that the footprint vectors give information about the access pattern of the loop and give an idea about which part of the array might be in the data cache after the execution. For example, if $\vec{v}^i = [1 \quad 0]^T$ and $\vec{v}^j = [0 \quad 1]^T$ and i is the outer loop, we know that, after the execution, only the last couple of rows can stay in the cache (if the cache capacity is small).

After determining the footprint vectors, we need to transform the code of the second process such that the resulting loop takes the best advantage of the contents of the cache. This can be achieved if we traverse the array (in the second process) along a footprint vector direction which is the opposite of the footprint vector direction of the first process. As an example, suppose that a process b has the following code:

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    e = e + (U[i][j]*U[i][j]);
```

Since from the first process (process a) we have $\vec{v}^i = [1 \quad 0]^T$ and $\vec{v}^j = [0 \quad 1]^T$, the loop of process b should be transformed such that the transformed loop has $\vec{v}^i = [-1 \quad 0]^T$ and $\vec{v}^j = [0 \quad -1]^T$. In other words, the array should be traversed from right to left and from bottom to top. This is so because such a traversal starts with the array elements that are most likely to be in the cache after process b has completed its execution. In order to restructure the loop in process b to obtain these footprint vectors, we build an equational system and solve it for the entries of T, the loop transformation matrix (see Section 2). In our current example, this system has the following form:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} T^{-1} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

To see this, recall that the footprint vector in the jth position is the jth column of the access matrix. Therefore, the transformed access matrix ($L' = LT^{-1}$) should have the desired footprint vectors as its columns. In this example, we obtain:

$$T^{-1} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \implies T = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}.$$

Note that this transformation corresponds to loop reversal [13] and results in the following transformed loop for the process b:

```
for (i=n-1; i ≥ 0; i--)
  for (j=n-1; j ≥ 0; j--)
    e = e + (U[i][j]*U[i][j]);
```

In general, in order to transform the loop belonging to a process b assuming that it will be scheduled after a process a, our approach first determines the footprint vectors for the loop in process a. It then computes a new set of footprint vectors (for the loop in b) such that if the loop in b is transformed to obtain these footprint vectors, most of the data in the cache (i.e., the contents after the execution of a) will be reused by b. In order to find the new footprint vectors, we simple negate the entries of the footprint vectors of a. To find the loop transformation matrix that realizes these new footprint vectors, we just build a new (desired) access matrix (for process b) from these vectors and solve the resulting system for the entries of T.

When we take into account transformations, computing $|S_{a,b}|$ for a given process pair (a, b) is done as follows. First, we try to determine the elements that remain in the cache after the process a's execution. To do this, we employ a binary search algorithm and the Omega library, and try to find an iteration point $K \in \mathcal{I}$ such that the elements accessed by iterations K, $K+1$, $K+2$, ..., K' (where K' in the last iteration) fill the cache. Then, using the Omega library and these iterations, we compute the array elements in the cache. As explained in this subsection, the code of the process b is transformed taking into account the access pattern of the code of the process a and is expected to use the elements that remain in the cache after a's execution. Thus, if the compiler is able to transform b to exploit the cache contents (that is, if data dependences allow such a loop transformation), we set $|S_{a,b}|$ to the contents of the cache after a's execution. If, on the other hand, such a loop transformation is not possible (e.g., due to data dependences), we conservatively set $|S_{a,b}|$ to 0. Note that while this is not a very accurate strategy, our experimentation shows that it performs very well in practice.

4. LOCALITY-CONSCIOUS SCHEDULING

As in the problem of evaluating inter-process data reuse, we solve the problem of determining a suitable schedule in two levels.

4.1 Scheduling Without Transformations

Having computed $|S_{a,b}|$ for all processes a and b, we need to select an order of execution for processes. We first build a data reuse table where each entry (a, b) gives the degree of reuse between process a and process b (see Figure 1). We then determine a schedule using the process dependence graph (PDG) and the entries in this table. Note that we can use any scheduling algorithm on PDG to determine a legal schedule. One such algorithm is list scheduling. However, since we want to exploit inter-process data reuse as much as possible, in selecting the next node (process) to schedule, we use the reuse information available in the data reuse table we built.

More specifically, suppose that we have just scheduled node v_a and we can schedule t nodes, v_{b_1}, v_{b_2}, ..., $v_{b_{t-1}}$, and v_{b_t}, immediately after v_a. In this case, we select the node by considering the entries (a, b_1), (a, b_2), ...,(a, b_{t-1}), and (a, b_t) in the data reuse table, and picking up the one with the largest value (i.e., the largest degree of reuse). Note that this approach is a greedy heuristic and is not guaranteed to generate the best result in every case. One particular problem with this approach is that it does not take into account the fact that after execution the last process the first process (in the schedule) will be executed again: so, the degree of reuse between the last and the first processes might also make a difference. However, if there are large number of processes, the impact of this last node-to-first node transition will be minimal.

4.2 Scheduling Using Transformations

When we take into account the possibility of transforming the source codes of the processes, the optimization strategy changes drastically. This is because in the previous case (i.e., the one without transformations) it was possible to separate the problem of determining degrees of reuse between processes from that of finding a schedule. That is, we can solve first the reuse degree determination problem and then the scheduling problem. However, when we consider loop transformations, such a separation is not possible due to the fact that the best loop transformation (hence, the maximum degree of reuse) for a given process depends on the process that immediately precedes it in the schedule. Therefore, selecting a good schedule and determining loop transformations should be handled together.

Recall that the transformation strategy discussed in Section 3.2 indicates that if we know that process b will (immediately) follow process a, we can transform process b's code to take advantage of the data that are left by process a in the cache after its execution. Based on this mechanism, we propose a branch-and-bound solution to the problem. While the worst-case complexity of this approach is exponential, its average complexity can be much less, making it a viable option in practice. In this approach, an alternative (potential) solution is not investigated further if its estimated cost is already larger than the minimum cost found so far. In determining and reducing the additional cost of scheduling a node (that is, adding a node to a partial schedule), we use the loop transformation strategy discussed in Section 3.2.

As an example, consider the decision tree fragment illustrated in Figure 3, where each node corresponds to a process and each path represents a partial solution (schedule). Let us focus on the leftmost branch in the tree. We see that this branch indicates the schedule: $a_1 - a_5 - a_2 - a_4 - a_3$. In obtaining this schedule, we proceed as follows. We first transform the code of a_5 considering the footprint vectors of a_1. After that, a_2 is transformed taking into account (the already transformed version of) a_5. Then, a_4 is transformed based on the new footprint vectors of a_2 and, finally, a_3 is transformed considering a_4.

Let us now focus on the partial solution $a_1 - a_5 - a_3$ in Figure 3.

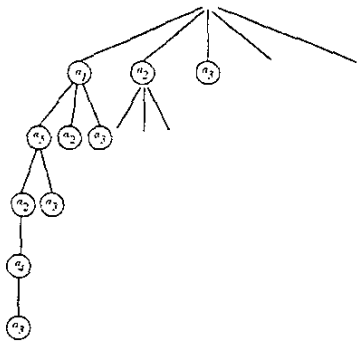

Figure 3: A decision tree fragment.

We define a *best degree of reuse* (BDR) of this partial solution as:

$$|S_{a_1,a_5}| + |S_{a_5,a_3}| + BEST(a_3, < a_2, a_4 >),$$

where $BEST(a_3, < a_2, a_4 >)$ is the best possible degree of reuse considering that a_3 is the node (process) that has just been scheduled and a_2 and a_4 are the nodes (processes) to be scheduled next. To compute $BEST$, we (optimistically) assume that the nodes to be scheduled next will have the best degree of reuses between them. So, the best degree of reuse for the partial solution given above has two components. The first component gives the degree of reuse of the partial solution itself, whereas the second component represents the best possible degree of reuse for the remaining nodes (i.e., the unscheduled ones). In a sense, the best degree of reuse is the best result that can be obtained when one starts with a partial solution (schedule). In general, we compute the best degree of reuse (BDR) of a partial schedule p as:

$$\begin{aligned} BDR(p) &= \text{degree of reuse}(p) \\ &\quad + BEST(\text{last}(p), \text{unscheduled nodes}). \end{aligned}$$

Note that, for a complete schedule p', its best degree of reuse (that is, $BDR(p')$) and its degree of reuse (that is, $DR(p')$ are the same.

Returning to the example in Figure 3, suppose that the branch-and-bound strategy has completed the analysis of the leftmost path and computed its degree of reuse, $DR(p)$, where p is $a_1 - a_5 - a_2 - a_4 - a_3$. Assume that we are now considering whether to explore the partial solution $p' = a_1 - a_5 - a_3$ further. In deciding that, we check whether

$$BDR(p') < DR(p)$$

hold. If it does, then we further explore this partial solution. Otherwise, we stop there and this partial solution is not explored further; instead, we move to another branch in the decision tree. During execution, our branch-and-bound algorithm keeps the current minimum cost and the current best schedule and decides whether to explore each subtree by comparing its cost (BDR) with the current minimum cost.

5. EXPERIMENTS

To test the effectiveness of our locality-conscious scheduling strategy, we performed experiments with array-based versions of two large, real-life embedded applications: *encr* and *usonic*. *encr* implements an algorithm for digital signature for security. It has two modules, each with eleven processes. The first module generates a cryptographically-secure digital signature for each outgoing packet in a network architecture. User requests and packet send operations are implemented as processes. The second module checks the digital signature attached to an incoming message. The main data structure used is an array of lists. The application code contains 335 C lines.

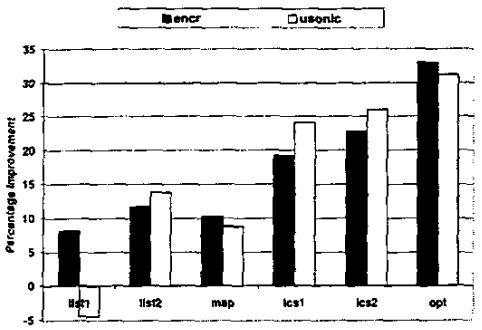

Figure 4: Results with small input size.

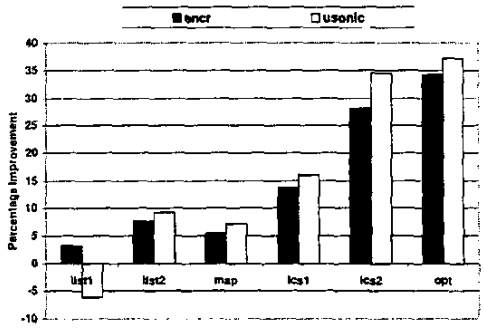

Figure 5: Results with large input size.

Our second application, *usonic*, is a feature-based object estimation algorithm. It stores a set of encoded objects. Given an image and a request, it extracts the potential objects of interest and compares them one-by-one to the objects stored in the database (which is also updated). It is written in C, consists of twelve processes, and contains 830 lines. For each application, we experimented with two different input sizes: small and large. With the small input size, most of the processes do not overflow data cache, whereas with the large input they do.

We performed experiments with seven different scheduling strategies. *original* is a list scheduling technique that selects the next node (process) to schedule randomly from among all schedulable nodes. *list1* is another list scheduling based strategy. In selecting the next node to schedule, it gives priority to the process with the shortest execution time. *list2* is also a list scheduling strategy; but, in selecting the node to schedule, it favors the node which has the maximum number of common arrays with the last node scheduled. *map* is a strategy that uses list scheduling; but, once the schedule has been determined, it maps the arrays onto memory locations such that the array used by neighboring processes (in the schedule) do not occupy same cache lines as much as possible (if they are not common arrays). *lcs1* and *lcs2* are the two layout-conscious scheduling strategies explored in this work. *lcs1* does not use any code transformation, whereas *lcs2* does. Finally, *opt* is an optimal strategy based on integer linear programming (ILP). Once the degrees of reuse have been found (as in *lcs2*), *opt* uses ILP to find the best schedule. The details of the *opt* and *map* strategies are omitted due to lack of space. All scheduling strategies are evaluated using a simple simulator. This simulator takes an input program (PDG), scheduling strategy, cache configuration, and cache simulator as parameters, and records the number of cache hits/misses and execution cycles. For cache simulations, we used a modified version of Dinero III [10].

We only present execution time results as cache hit/miss trends are very similar to execution time trends. Figures 4 and 5 give (for the small and large input sizes, respectively) the percentage improvements in execution time with respect to the *original* version. The cache used is 16KB, two-way associative with a line size of 32 bytes. Based on these results, we make the following observations. First, we see that both *lcs1* and *lcs2* versions perform better than the rest (except the *opt* version), indicating that locality-conscious scheduling is successful in practice. Second, the performance of *lcs2* is close to that of *opt*, meaning that a list scheduling based strategy makes sense provided that the cost metrics used are accurate enough. Third, the difference between *lcs2* and *lcs1* is much larger with the large input. That is, when the input size is increased, using code transformations for enhancing inter-process reuse becomes more critical. Fourth, the *map* version does not perform very well, mainly due to the fact that it is very difficult to map arrays in memory in a conflict-free manner. The results in Figures 4 and 5 are obtained using the most frequently occurring footprint vector as the summary footprint vector. When we use vector combination instead, we observed a 3.4% (4.1%) additional improvement in *encr* (*usonic*).

6. CONCLUSIONS AND FUTURE WORK

We have presented a locality-conscious process scheduling strategy and shown that taking into account inter-process data reuse during process scheduling can make large differences in execution time. We have also shown that in cases where the sizes of datasets manipulated by processes are very large, compiler-based transformations might improve inter-process data reuse. We plan to extend this work by relaxing the assumptions we made. We will also perform more experiments with different media applications with varying input sizes.

7. REFERENCES

[1] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design.* Kluwer Academic Publishers, June, 1998.

[2] M. Chiodo et al. Hardware-software codesign of embedded systems. *IEEE Micro*, 14, No 4, August 1994, pp. 26–36.

[3] G. De Micheli and R. K. Gupta. Hardware-software codesign. *Proceedings of the IEEE*, 85, No 3. (March 1997):349–365.

[4] R. Ernst. Codesign of embedded systems: status and trends. *IEEE Design and Test of Computers*, 15, No 2, (April-June 1998):45–54.

[5] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–51, 1991.

[6] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proc. the ACM Design Automation Conference*, Los Algeles, CA, 2000.

[7] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. The Omega Library interface guide. *Technical Report CS-TR-3445*, CS Dept., University of Maryland, March 1995.

[8] C-G. Lee et al. Analysis of cache related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6), June 1998.

[9] Y. Li and W. Wolfe. A task-level hierarchical memory model for system synthesis of multiprocessors. *IEEE Transactions on CAD*, 18(10), October 1999, pp. 1405–1417.

[10] WARTS: Wisconsin Architectural Research Tool Set. http://www.cs.wisc.edu/~larus/warts.html

[11] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[12] W. Wolfe. *Computers as Components: Principles of Embedded Computing System Design*,Morgan Kaufmann Publishers, 2001.

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

[14] A. Wolfe. Software-based cache partitioning for real-time applications. In *Proc. the Third International Workshop on Responsive Computer Systems*, September 1993.