Compiler-Based Code Partitioning for Intelligent Embedded Disk Processing

Guilin Chen[1], Guangyu Chen[1], M. Kandemir[1], and A. Nadgir[1]

The Pennsylvania State University, University Park, PA 16802, USA
{guilchen,gchen,kandemir,nadgir}@cse.psu.edu

Abstract. Recent trends indicate that system intelligence is moving from main computational units to peripherals. In particular, several studies show the feasibility of building an intelligent disk architecture by executing some parts of the application code on an embedded processor attached to the disk system. This paper focuses on such an architecture and addresses the problem of what parts of the application code should be executed on the embedded processor attached to the disk system. Our focus is on image and video processing applications where large data sets (mostly arrays) need to be processed. To decide the work division between the disk system and the host system, we use an optimizing compiler to identify computations that exhibit a *filtering characteristic;* i.e., their output data sets are much smaller than their input data sets. By performing such computations on the disk, we reduce the data volume that need to be communicated from the disk to the host system substantially. Our experimental results show significant improvements in execution cycles of six applications.

1 Introduction

Recent years have witnessed several efforts towards making the disk storage system more intelligent by exploiting available computing power within the disk subsystem. A common characteristic of these proposals (e.g., active disks [22, 1], intelligent disks [11], smart disks [15]) is to use computing power at the disk (provided by an embedded processor attached to the disk), to perform some filtering type of computations on the storage device itself. For example, [18] demonstrates how several database operations can be performed by the embedded processor attached to the storage device. A similar project in IBM [9, 8] attempts to embed processing power near the data (e.g., on the disk adapter) to handle general purpose processing offloaded from the host system. Thus, an intelligent disk-based computation can significantly reduce the demand on the communication bandwidth between the storage device and the rest of the system. Uysal et al propose an active disk architecture by allowing more powerful on-disk processing and large on-disk memory [22]. To address the software design and implementation for active disks, Acharya et al describe a stream-based programming model, whereby host-resident code interacts with disk-resident code using streams [1]. The active disk concept proposed by Riedel et al [18] helps us investigate the behavior of scan-based algorithms for databases, nearest neighbor search, frequent sets, and edge detection of images on such architectures. The authors use these applications to show performance improvements brought by active disks over conventional architectures. Sivathanu et al

[21] propose the concept of semantically-smart disk system, wherein the disk system obtains from the file system information about its on-disk data structures and policies. It then exploits this information by transparently improving performance.

Most of these prior studies focus on application programming model and operating system (OS) support for intelligent disk architectures. While this support is critical for the successful deployment of such architectures, for a large application, it would be very difficult for an average programmer to decide what to execute on the embedded processor on the disk and what to execute on the host system. In this paper, we address this important problem and propose a compiler-based strategy that automatically divides an application between the disk system (the embedded processor) and the host system. Our focus is on image and video applications where large data sets (arrays) need to be processed. To decide the work division between the disk system and the host system, we identify computations that exhibit a *filtering characteristic;* i.e., their output data sets are much smaller than their input data sets. By performing such computations on the disk, we reduce the data volume that need to be communicated from the disk to the host system significantly. In fact, our experiments with several applications reveal communication volume reductions around 50%. Obviously, such a reduction in communication volume between the disk and the host system helps reduce power consumption and enhances overall system performance.

It should be emphasized that such intelligent disk architectures are actually being built by disk drive and chip manufacturers. For example, Infineon markets a chip called TriCore that includes a 100 MHz micro-controller, up to 2MB of main memory, and some custom logic that implements disk drive-specific functions [18]. Considering the drops in costs of embedded processors and the growing demand for data-intensive computing, we can expect that such architectures will be more prevalent in the future. It should also be mentioned that while we present our approach that exploits filtering characteristic of a computation to reduce communication demands in the context of a disk-host pair, the compiler analysis presented here is general enough to be employed in other circumstances where filtering data before communication is desirable (e.g., in a sensor network based environment where sensors process the collected data before passing it to a central base station).

2 Architecture and Programming Model

The architecture we focus on this study has two major components: host system and disk system. The host system is the unit where computations are normally performed. The disk system is the storage subsystem that consists of a disk (which might be a RAID) and an embedded processor which can be used to perform some of the computation that would normally be performed by the host system (i.e., a system without an embedded processor on the disk would perform all computations in the host system). A sketch of the architecture considered in this paper is given in Figure 1.

To make use of the embedded processor on the disk, we need instructions (or compiler directives) to map some application code portions to the disk system. In this paper, we assume the existence of two compiler directives, called begin{map} and end{map}, that enclose a code portion which will be executed on the disk system.

Note that, in a given application, these directives can be used a number of times. Also, the code portions (fragments) that can be enclosed by these directives can be of different granularities (e.g., a loop, a loop nest, or an entire procedure). However, since our focus in this work is on array-intensive applications, we work on a loop nest granularity. In other words, for each nest of a given application, our approach decides whether to execute that nest on the host system or on the disk system. In this work, the begin{map} and end{map} directives are automatically inserted in the application code by the compiler.

It should be noted that in this architecture a given data set can be in memory or in the disk. We use the term disk-resident to indicate that the data set (array) in question resides in the disk system. To enable efficient compiler analysis, we assume that the disk-resident arrays are annotated using a special compiler directive. Note that the data transfers between the disk-resident and memory-resident data sets are explicit. That is, to copy a disk-resident data set (or a portion of it) to a memory-resident data set, one needs to perform an explicit file operation. However, to make our presentation clear, in the code fragments considered in this paper, we mix disk-resident and memory-resident data set (array) accesses, assuming implicitly that each access to a disk-resident array involves a file operation. Our approach can operate with cases where only some of the arrays are disk-resident and also with cases where all of the arrays are disk-resident. It should be mentioned that user-inserted compiler directives have been employed in the past in the context of parallel programming to govern data distributions across memories of multiple processors [13].

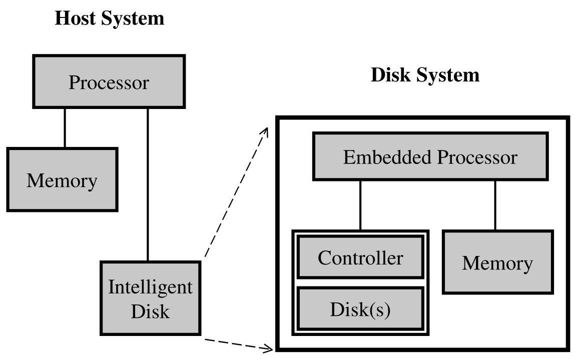

Fig. 1. The sketch of the architecture considered in this work.

3 Work Division

An important issue that need to be addressed for extracting the maximum benefit from our storage architecture is to divide application execution between the host system and the disk system. To accomplish computation partitioning between the disk system and the host (also called work division), our compiler analyzes the entire application to extract data access pattern. It then inserts begin{map} and end{map} calls in the code to perform work division. In our implementation, a code fragment is mapped on to the disk system if it satisfies the following two criteria:

- It should perform input/output (I/O). While the embedded processor in our architecture can be exploited for performing non-I/O related functionalities as well, in this study, we consider only I/O-intensive code fragments for potential candidates to be executed in the disk system. Since we require disk-resident data sets to be explicitly identified by the programmer, we can easily check whether a given computation performs I/O or not (i.e., we just check whether it involves a disk-resident data set).
- It should exhibit a filtering characteristic. A code fragment is said to exhibit a filtering characteristic if the size of its input data sets (arrays) is much larger than its output data sets (arrays). As an example, consider the following code fragment that consists of two separate nests (written in a pseudo-language):

```
for I = 2..N-1
 for J = 2..N-1
  V[I][J] = 0.25 * (U[I][J-1] + U[I][J+1]
                    + U[I-1][J] + U[I+1][J])
for I = 1..N
 for J = 1..N
  for K = 2..N-1
   X[I][J] = 0.33 * (W[I][J][K-1] + W[I][J][K]
                     + W[I][J][K+1])
```

  Assuming that arrays U and W are disk-resident, the first nest above does not have any filtering characteristic since it takes a two-dimensional array (U) and generates another two-dimensional array (V). In contrast, the second nest exhibits filtering. This is because it takes a three-dimensional array (W) and generates a two-dimensional array (X). Therefore, it is a better candidate to be executed on the disk system. It should be noticed that, if we do not execute this second nest on the disk system, it needs to be executed in the host system. But, in this case, to perform the required computation, we need to transfer the entire data set W from the disk to the host system, resulting in tremendous network traffic. This obviously will eat up lots of execution cycles and waste the storage bandwidth (it also increases system-wide energy consumption). This is exactly the overhead that we want to eliminate. Instead, if we can execute this nest on the disk system, we need to transfer only the resulting data set (X) to the host system (so that it can be used by the rest of the application). In this way, data is filtered in the disk system before it is shifted to the host system, thereby leading to an improvement in overall performance. The only drawback of performing the computation on the disk system (instead of the host) is that it will take longer time as the embedded processor on the disk is typically less powerful than the host processor.

3.1 Detecting the Filtering Characteristic

In this work, we experiment with two different strategies for detecting whether or not a given computation exhibits a filtering characteristic. The first strategy (called Strategy I) is easy to implement and (as will be shown later in the paper) generates very good results in practice. It checks (for each array) the number of dimensions and their extents (i.e., dimension sizes). Let us consider the following loop nest and the assignment statement shown.

```
for i1 = L1..U1
 for i2 = L2..U2
  ...
   for is = Ls..Us
     U[f1][f2]...[fn]  = ... V[g1][g2]...[gm] ...
```

Here, we assume that `f1`, `f2`, ..., `fn`, `g1`, `g2`, ...,`gm` are the subscript expressions (array index functions), and each `fi` and `gj` is an affine function of loop indices `i1`, `i2`, ..., `is` and loop-independent variables. Assuming further that arrays `U` (`n`-dimensional) and `V` (`m`-dimensional) are declared as `type U[N1][N2]...[Nn], V[M1][M2]...[Mm]`, where `type` be any (data) type such as integer or float, Strategy I decides that the assignment statement in the loop shown above exhibits a filtering characteristic if:

```
c x N1 x N2 x ... x Nn < M1 x M2 x ... x Mm.
```

In this last expression, `c` is a constant to make sure that the difference in the sizes of input (right-hand-side) and output (left-hand-side) arrays is large enough so that shifting the computation (the statement) to the disk system will be really beneficial (note that if `c`=1, this corresponds to the informal description of the concept of "exhibiting a filtering characteristic" that we have been using so far). In most of the cases (of array-based applications) encountered in practice, it is possible to check the above condition statically (at compile-time). In cases where this is not possible, we have at least two choices. First, we can employ profile data (e.g., by instrumenting the code) to see whether the condition holds for typical data sets. Second, we can insert a conditional statement (if-statement) into the code that chooses between performing computation on the host side and performing it on the disk side, depending on the outcome of the condition. It is to be noted that selecting a suitable `c` value is critical. This is because a small `c` value can force aggressive computation mapping to the disk system. This in turn can result in some unsuitable computation being mapped to the embedded processor, thereby reducing overall performance. On the other hand, a very large `c` value can be overly conservative and can result in a code mapping that does not exercise the embedded processor at all. Since the best value for `c` is both application and architecture dependent, it is not possible to determine an optimal value statically (compile-time). As a result, in this paper, we experimented with different `c` values (instead of fixing it at a specific value). If, in a given loop, there is at least one statement that exhibits filtering characteristic, we mark the entire loop to be executed on the disk system (i.e., we assume that the loop has filtering characteristic). Later in the paper, we demonstrate how loop transformations can be used for improving the effectiveness of our optimization strategy.

As an example, let us consider again the code example shown above in Section 3 (which consists of two separate nests). Assuming that all array dimensions are of the same size (extent), using the approach summarized in the previous paragraph, one can easily see that only the second nest is identified to be executed on the disk system (assuming c = 1). While it might be possible to have more elaborate strategies for identifying the loops that need to be mapped to the disks system, as the experimental results (presented later) show, Strategy I performs well in practice.

Our second strategy (called Strategy II) is more sophisticated but can also be expected to generate better results than Strategy I. Considering the loop nest and the assignment statement shown above, this strategy decides that the assignment statement exhibits a filtering characteristic if:

```
c x G{U[f1][f2]...[fn]} < G{V[g1][g2]...[gm]}
```

where c is the same as described earlier and G{E} gives the number of distinct array elements accessed by array reference E. In other words, instead of just checking the bounds of the arrays involved in the computation, Strategy II checks the actual number of elements accessed. Consequently, in general, it can be more accurate than the first strategy (since not all the loops access all the elements of the arrays they manipulate). The drawback is that determining the exact number of elements accessed by an affine expression is a costly operation [19, 6]. In this paper, we adopt the first strategy as our default strategy; but, we also perform experiments with the second strategy to demonstrate its potential in some applications. In implementing Strategy II, we represent the set to be counted using the Presburger formulas and use the technique proposed in [17].

3.2 Reducing Communication Between the Host System and the Disk System

It should be clear that mapping large code fragments to the disk system is preferred to mapping smaller ones as the former implies less communication between the host code fragments and the fragments mapped to the disk system. To determine whether two neighboring code fragments, say Frag1 and Frag2, should be mapped to the disk system as a whole or not, we adopt the following strategy. Suppose that Frag1 generates an output dataset X that will subsequently be used by Frag2. If X is also requested by the host code fragment (in addition to Frag2), then we need to transfer X to the host system. In this case, Frag1 and Frag2 are treated independently (i.e., they are not combined). On the other hand, if Frag2 is the only consumer of X, then these two fragments can be combined together (i.e., they can be mapped to the disk using the same begin{map}-end{map} construct; no communication is necessary when execution moves from Frag1 to Frag2, or vice versa), and X does not need to be transferred to the host system at all (saving bandwidth as well as latency). Many array-based applications exhibit such producer-consumer relationships. In particular, in array-intensive applications, given two nests, an optimizing compiler can, test whether they should be mapped together or not.

Our current implementation uses data-flow analysis for this purpose. Data-flow analysis is a program analysis technique that is mainly used to collect information about how data flows through program statements/blocks [16]. In our context, we use data-flow analysis to determine the the nests that will be mapped to the disk system together. Our approach can be summarized as follows. First, using the strategy explained above (Section 3.1), we determine the set of nests that should be mapped to the disk system. This set represents the minimum set of nests to be mapped to the disk. After that, using the strategy explained in the previous paragraph, we determine the additional nests to be mapped to the disk. These are typically the nests that are the only customers for the data generated by a nest from the set determined in the first step. After this process, each loop nest in the application is assigned to be executed either on the disk or on the host system, and the corresponding begin{map} and end{map} directives are inserted in the code. We omit the formal description of our data-flow algorithm due to space concerns.

4 Parallel Processing on the Disk System

In our architecture considered so far, we have assumed only a single embedded processor. However, in many array-intensive applications, the computations mapped to the disk system has some degree of loop-level parallelism. That is, the loop iterations can be executed in parallel. Therefore, it makes sense to consider the compiler support for a more aggressive architecture that consists of multiple embedded processors on the disk system. This would lead to the following additional constraint (in addition to the two criteria described earlier in Section 3) to map a code fragment onto the disk system:

– The code fragment should take advantage of the parallel embedded processors on the disk system. In other words, the code portions mapped to the disk system should be parallelizable. This parallelization can be achieved in two ways. First, for array-intensive applications (which is the focus of this work), the compiler can analyze data dependences between the loop iterations [16] and can detect whether the loop can be parallelized. Second, for other types of codes (e.g., those that make heavy use of pointer arithmetic), the user can annotate parallel code fragments, and this can help our compiler in deciding which code portions must be mapped to the disk system. As an example of the first type of scenario, the second nest of the fragment shown in Section 3 exhibits loop level parallelism. More specifically, `I` and `J` loops can be parallelized across the embedded processors on the disk. It should also be noted that sometimes it might be beneficial to relax our requirements for mapping data on to the disk system, and still take advantage of our storage architecture. For example, in some cases, the I/O portion of the application code may not be parallelizable; but, mapping it to the disk system can lead to large reductions in communication volume due to filtering type of computation on disk-resident data. Similarly, in some cases, there may not be any data filtering activity, but we may have large amount of I/O parallelism. Again, mapping this I/O (and the associated computation) to the disk system can reduce I/O and execution time.

In our current implementation, we can work with two different styles of parallelism. First, we can allow the programmer to annotate the loops in the program to execute

Parameter	**Value**
Host Processor	
Functional Units	4 integer ALUs 1 integer multiplier/divider 4 FP ALUs 1 FP multiplier/divider
LSQ Size	8 Instructions
RUU Size	16 Instructions
Fetch Width	4 instructions/cycle
Decode Width	4 instructions/cycle
Issue Width	4 instructions/cycle
Commit Width	4 instructions/cycle
Fetch Queue Size	4 instructions
Clock	1 GHz
Embedded Processor	
Functional Units	2 integer ALUs 1 integer multiplier/divider 2 FP ALU/multiplier/divider
Fetch/Decode/Issue/Commit	1 instruction/cycle
Clock	200 MHz
Cache and Memory Hierarchy	
L1 Instruction Cache	16KB, 1-way, 32 byte blocks 1 cycle latency
L1 Data Cache	16KB, 4-way, 64 byte blocks 1 cycle latency
L2 (only Host)	256K unified, 4-way 64 byte blocks 6 cycle latency
Main Memory (Host)	128MB, 150 cycle latency
Main Memory (Embedded)	16MB, 75 cycle latency
Branch Logic (only Host)	
Predictor	combined, bimodal 2KB table two-level 1KB table 8 bit history
BTB	512 entry, 4-way
Misprediction Penalty	45 cycles
Storage System and Interconnects	
Stripe Size	16 KB
RAID Level	5
Individual Disk Capacity	33.6 GB
Disk Cache Size	4 MB
Disk Rotation Speed	10000 RPM
Disk-Arm Scheduling	Elevator
Bus Type	Ultra-3 SCSI
Embedded Processor Communication	TB3 switches (50.9 MBps)
Host Processor Communication	155 MBps
Interconnection Network (between host & disk)	160 MB/sec

Table 1. Default simulation parameters used in our experiments.

parallel. Second, we have developed a strategy that parallelizes a sequential application based on data reuse analysis. The approach used tries to put as much data reuse as possible into innermost loop positions, hence leaves dependence-free outer loops to be parallelized. The details of this approach is beyond the scope of this paper and can be found elsewhere [10]. In the next section, we experimentally (quantitatively) evaluate our approach to see whether it improves performance in practice.

5 Experiments

5.1 Simulation Environment

We designed and implemented a custom simulation environment to perform our experiments. This environment can simulate systems with different number of host processors, disks, and embedded processors. Our simulator uses DiskSim [7] for simulating the disk behavior. DiskSim is an accurate and highly-configurable disk system simulator developed at the University of Michigan and enhanced at CMU to support research into various aspects of storage subsystem architecture. It includes modules for most secondary storage components of interest, including device drivers, buses, controllers, adapters, and disk drives. The detailed disk module employed in DiskSim has been carefully validated against ten different disk models from five different manufacturers. The accuracy demonstrated exceeds that of any other publicly-available disk simulator [7]. The simulation of the host processor(s) and embedded processor(s) have been performed using Simplescalar [3] infrastructure. To simulate communication between processors, we adopted a simple strategy based on the number of communication messages and the available bandwidth. The default simulation parameters used in the experiments for processors, disk and communication subsystems are listed in Table 1. Unless stated otherwise, all experimental results to be presented have been obtained using the simulation parameters in this table. It is to be noted that the parameters given in the cache and memory hierarchy part are the same for both the host and embedded processors; the cases where they differ are specified explicitly.

We conducted experiments with two configurations: one without the embedded processor attached to the disk, and one with the embedded processor. The first configuration is called the *base configuration* and has a host processor of 1GHz with a 128MB memory space. In the second configuration, the host processor speed is again 1GHz, but we also have a 200MHz embedded processor with a 16MB memory space (Texas Instruments' C27x series, for example, has this memory capacity) attached to the disk system. The I/O interconnect between the disk system and the host system is assumed to be 160 MB/s (a reasonable value for a typical disk-based architecture). When we have multiple host processors (or embedded processors), they communicate with each other using respective communication networks.

Figure 2 illustrates how the simulations have been performed. First, the application code is divided between the host system and the disk system (as discussed earlier in detail). Then, the host program is simulated using the CPU simulator and the communication simulator. Similarly, the disk program (i.e., the code portion mapped to the disk system) is simulated using the CPU simulator, the communication simulator, and the disk simulator. The last phase collects the statistics and combines them.

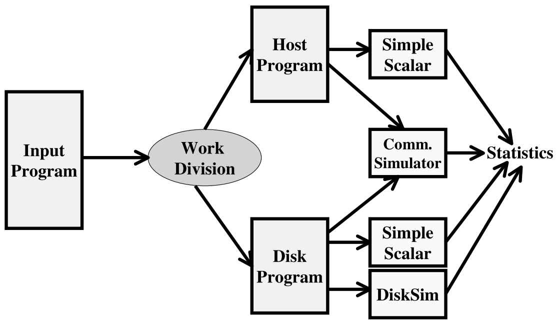

Fig. 2. Simulation process.

Benchmark	Brief Description	Total Dataset Size	Execution Cycles				Filtering Statements	
			Cycles	I/O	Computation	Communication	Strategy I	Strategy II
Feature	Feature Extraction	11.72MB	12582216517	41.4%	28.8%	29.8%	26.8%	26.8%
ImgComp	Image Compression	8.21MB	9763488176	35.5%	24.1%	40.4%	41.3%	47.7%
Restore	Image Restoration	35.40MB	46055704201	39.1%	27.2%	33.7%	36.6%	36.6%
SMT	Video Smoothing	17.06MB	18407842088	48.2%	20.6%	31.2%	39.0%	39.0%
T-Image	Crowd Management with Imaging	16.05MB	16342777094	32.2%	26.6%	41.2%	44.3%	44.3%
Vehicle-V	Vehicle Tracking and Classification	23.88MB	34191529330	44.4%	18.9%	36.7%	21.5%	26.9%

Table 2. Benchmark codes used in the experiments.

5.2 Benchmarks and Code Versions

To evaluate the performance of our strategy, we conducted experiments with six array-intensive benchmark programs.

- Feature: This is a speech processing program that implements perceptual linear prediction (PLP). PLP is based on the short-term spectrum of speech. In contrast to pure linear predictive analysis of speech, perceptual linear prediction (PLP) modifies the short-term spectrum of the speech by several psychophysically based transformations.
- ImgComp: This code implements a wavelet transform-based image coder for gray-scale images. While the coder itself is not very sophisticated, each individual piece of the transform coder has been chosen for high performance.
- Restore: This is a high order iterative method for image restoration. As compared to pure iterative methods, it is much faster and converges after two dozens of iterations.
- SMT: This code implements a video smoothing algorithm using temporal multiplexing. The algorithm smoothes out the rate variability of the data transmission from a server to a client so that the network utilization can be improved.
- T-Image: This program controls outputs of several cameras connected to a single display. It implements a simple automatic video image processing system which outputs statistics such as detection of intrusion in forbidden areas and detection of abnormal lack of movement or counterflow movements.

– Vehicle-V: This code implements an algorithm that employs a predictive Kalman filter to track motion through occlusions (2D rigid motion). It also calculates the gradient of the error to adjust estimation.

The number of C lines of the sources of these applications range from 465 to 3,128. Their important characteristics are given in Table 2. The third column shows the total size of the disk resident data manipulated by each application. The fourth column gives the execution cycles for the original codes on our base configuration (i.e., without the embedded processor on the disk system). The next three columns give the distribution (breakdown) of execution cycles into three categories: the cycles spent in I/O; the cycles spent in computation (on the host); and the cycles spent in communication between the host system and the disk system. We see that a significant number of cycles spent in communication, which means that minimizing communication volume can bring large performance benefits in practice. The reason that we used these specific applications is that they were available to us from University of Manchester, and that they represent a good mix of real-life applications as far as the filtering capability and I/O parallelism is concerned (that is, some of them do not have much filtering (I/O parallelism), whereas the others have). This last point is indicated in the last two columns of Table 2, where we give the percentage of statements with the filtering characteristic for the two strategies (Strategy I and Strategy II) explained in Section 3.1. One can see from these last two columns that our benchmarks include codes with low filtering opportunity (e.g., Vehicle-V) as well as codes with high filtering opportunity (e.g., T-Image). We also see from these two columns that using Strategy I (which is based on array sizes) and Strategy II (which is based on the number of distinct elements accessed) results in, respectively, 34.9% and 36.8% of the statements being identified as exhibiting the filtering characteristic. This shows that there exists scope to perform intelligent computation on the disk system.

For each application, we used two different code versions. The *base version* is the one that runs on the architecture without embedded processor (i.e., the base configuration), and the *optimized version* is the one that runs on the architecture with embedded processor. To evaluate the impact of the number of processors, we also performed experiments with different number of host and embedded processors. All the results presented in this section are *normalized* with respect to *the base version with a single host processor* (i.e., the base configuration). Unless stated otherwise, no loop distribution is applied in the optimized version, Strategy-I is used, and the c value (see Section 3) is set to the largest dimension size of the largest array in the application. More specifically, the default c values for Feature, ImgComp, Restore, SMT, T-Image, and Vehicle-V are 1000, 512, 712, 256, 256, and 512, respectively. All code modifications (when loop distribution is employed) have been automated within the SUIF (Stanford University Intermediate Format) infrastructure from the Stanford University [2].

5.3 Results

The graph given in Figure 3 shows the normalized overall execution cycles for both original and optimized versions. All bars are normalized with respect to the first bar, which gives, for each application, the distribution of execution cycles, and divided into three parts: (i) the time spent in computation (on the host or on the disk system); (ii) the time spent in communication (between the host and the disk); and (iii) the time spent in I/O on the disk. The second bar gives the execution cycle breakdown for the optimized version when a single embedded processor is used in the disk system. When we compare these first two bars, we observe that our compiler-based approach reduces the execution cycles spent in communication by 41.2% on the average. This results in a 10.1% reduction, on the average, on overall execution cycles.

The third bar for each application in Figure 3 gives the reduction in execution time when the number of embedded processors is increased to 4 (each is 200MHz). One can clearly see that this brings large reductions in the I/O time as well as the computation time spent by the embedded processors. This is due to exploiting parallelism on the disk system. As a result, we obtain a 49.1% execution cycle reduction on the average across all six applications. What this means is that the code portions shifted to the disk system take advantage of parallelism to a large extent. One might argue that increasing the number of host processors could also bring comparable benefits. The fourth bar in Figure 3 for each application gives the normalized execution cycles when the number of host processors on the host system is increased to 4, while keeping the number of embedded processors at 1. When we compare the third and the fourth bars, we see that the (overall) execution time reductions brought by increasing the number of embedded processors are, in general, higher than those obtained by increasing the number of host processors. In other words, using three additional (cheap—200MHz) embedded processors on the disk is more beneficial than employing three additional (powerful—1GHz) processors on the host. This is mainly because embedded processors can reduce I/O, computation, and communication times, whereas the host processor can reduce only I/O and computation times. Although the reductions in I/O/computation times due to increased number of host processors are higher than those due to increased number of embedded processors (as host processors are faster), this effect is overshadowed by the decrease in communication cycles as a result of increased number of embedded processors.

It is also conceivable that in the future the embedded processors will become very powerful, and it might be possible to put such powerful processors on the disk system. To quantify the benefits that could come from such systems, the last bar for each application in Figure 3 gives the normalized execution cycles, when 4 powerful (1GHz) embedded processors are used on the disk system (with only one host processors). These results clearly show that employing powerful processors on the disk system (rather than on the host system) is much more beneficial, reducing the overall execution cycles by 59.8% on an average.

To study the scalability of parallel processing on the disk system, we also conducted further experiments with different number of embedded processors (in conjunction with a single host processor). The results normalized with respect to the original version (with one host processor) are shown in Figure 4. We can clearly see that increasing the number of embedded processors generates good scalability; that is, where available, we are able to take advantage of the large number of embedded processors. For example, when we have 16 embedded processors, the average reduction in overall execution cycles is 67% across all benchmarks. It should also be observed that the additional benefits

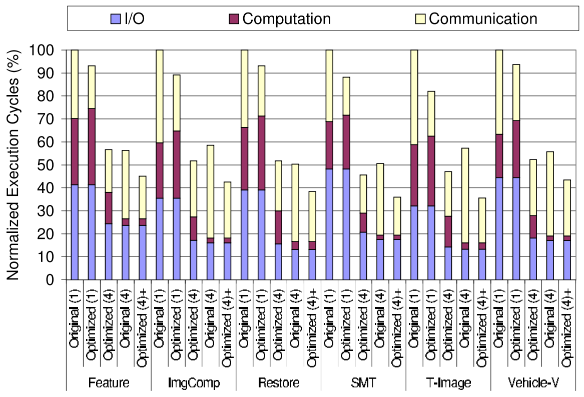

Fig. 3. Normalized execution cycles for different versions.

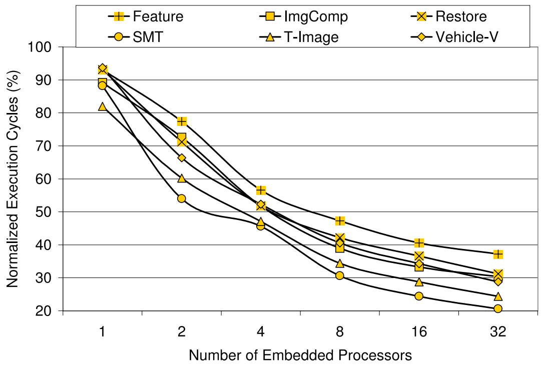

Fig. 4. Normalized execution cycles with different number of embedded processors.

of our approach gets reduced as one increases the number of processors to large values as a result of the contention on the disk system.

6 Concluding Remarks

Intelligent disk systems with large storage capacities and fast interconnects are expected to become prevalent in the next decade. This is due to the trends that try to bring computation to where data resides (instead of more traditional approach where the data is brought into where computation is normally executed). An important problem that needs to be addressed in such architectures is how to divide the computation between the disk system (embedded processor) and the host system (processor). This paper has proposed and evaluated a compiler-based work division (computation partitioning) strategy for array-intensive applications. Our strategy is based on the idea that the computations (loop nests) that filters their input data sets should be mapped on to the disk system. The experimental results with six application codes have indicated that the proposed approach is very successful in practice.

References

1. A. Acharya, M. Uysal and J. Saltz, ”Active Disks: Programming Model, Algorithms and Evaluation”. In Proc. the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.
2. http://suif.stanford.edu/
3. D. C. Burger and T. M. Austin. ”The SimpleScalar Toolset”, Version 2.0, Technical Report 1342, Dept. of Computer Science, UW, June, 1997.
4. R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. Anderson. ”Data Distribution Support on Distributed-Shared Memory Multiprocessors.” In Proc. Programming Language Design and Implementation, Las Vegas, NV, 1997.
5. A. Chandrakasan, W. J. Bowhill, and F. Fox. ”Design of High-Performance Microprocessor Circuits,” IEEE Press, 2001.
6. P. Clauss. ”Counting Solutions to Linear and Nonlinear Constraints through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs”. In Proc. the 10th International Conference on Supercomputing, pp. 278–285, May 25–28, 1996, PA.
7. G. Ganger. ”System-Oriented Evaluation of I/O Subsystem Performance”, Technical Report CMU-TR-243-95, University of Michigan, 1995.
8. W. Hsu, A. Smith, and H. Young, ”Projecting the Performance of Decision Support Workloads on Systems with Smart Storage (SmartSTOR)”. Report No. UCB/CSD–99–1057, 1999.
9. IBM Automatic Locality-Improving Storage (ALIS). http://www.almaden.ibm.com/cs/storagesystems/alis/index.html.
10. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. ”Improving Locality Using Loop and Data Transformations in an Integrated Framework.” In Proc. International Symposium on Microarchitecture, Dallas, TX, December 1998.
11. K. Keeton, D. Patterson and J. Hellerstein, ”A Case for Intelligent Disks (IDISKs)”. In SIGMOD Record, 27(3), 1998.
12. I. Kodukula, N. Ahmed, and K. Pingali. ”Data-centric multi-level blocking.” In Proc. SIGPLAN Conf. Programming Language Design and Implementation, June 1997.
13. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. ”The High-Performance Fortran Handbook”, MIT Press, Cambridge, MA, 1994.
14. K. McKinley, S. Carr, and C.W. Tseng. ”Improving Data Locality with Loop Transformations.” ACM Transactions on Programming Languages and Systems, 1996.
15. G. Memik, M. Kandemir and A. Choudhary, ”Design and Evaluation of Smart Disk Architecture for DSS Commercial Workloads”. In Proc. International Conference on Parallel Processing, September 2000.
16. S. S. Muchnick. ”Advanced Compiler Design Implementation.” Morgan Kaufmann Publishers, San Francisco, California, 1997.
17. W. Pugh ”Counting Solutions to Presburger Formulas: How and Why”, In Proc. the ACM Conference on Programming Language Design and Implementation 1994, Orlando, Florida.
18. E. Riedel, C. Faloutsos, G. Gibson and D. Nagle, ”Active Disks for Large-Scale Data Processing”. IEEE Computer, June 2001, pp. 68–74.
19. A. Schrijver. ”Theory of Linear and Integer Programming”, John Wiley and Sons, Inc., New York, NY, 1986.

20. S. Singhai and K. S. McKinley. ”A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. The Computer Journal”, 40(6):340–355, 1999.
21. M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, ”Semantically-Smart Disks Systems,” Technical Report 1445, Computer Sciences Department, UW, Madison, 2002.
22. M. Uysal, A. Acharya and J. Saltz, ”Evaluation of Active Disks for Decision Support Databases”. In Proc. International Conference on High Performance Computing Architecture, January 2000.