The Pipeline Decomposition Tree: An Analysis Tool For Multiprocessor Implementation Of Image Processing Applications

Dong-Ik Ko
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, 20742, USA.
dik@eng.umd.edu

Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, 20742, USA.
ssb@eng.umd.edu

ABSTRACT

Modern embedded systems for image processing involve increasingly complex levels of functionality under real-time and resource-related constraints. As this complexity increases, the application of single-chip multiprocessor technology is attractive. To address the challenges of mapping image processing applications onto embedded multiprocessor platforms, this paper presents a novel data structure called the pipeline decomposition tree (PDT), and an associated scheduling framework, which we refer to as PDT scheduling. PDT scheduling exploits both heterogeneous data parallelism and task-level parallelism, which are important considerations for scheduling image processing applications. This paper develops the PDT representation for system synthesis, and presents methods using the PDT to derive customized pipelined architectures that are streamlined for the given implementation constraints.

Categories and Subject Descriptors

C.3 [Real-time embedded systems]

General Terms

Design

Keywords

Design space exploration, system-level models, multiprocessor scheduling.

1. INTRODUCTION

The proliferation of embedded systems that involve image processing, such as digital cameras and video-conferencing systems, exhibits trends towards the integration of multiple image processing operations to provide diverse functionalities, and the application of embedded multiprocessor technology to provide the required performance. This paper presents a novel data structure called the *pipeline decomposition tree* (*PDT*), and an associated scheduling framework, which we refer to as *PDT scheduling*, for mapping image processing applications onto embedded multiprocessor systems. PDT scheduling is based on a model of the target implementation as a coarse-grained (task-level), pipelined architecture. PDT scheduling spreads functional operations over the underlying pipeline through construction and iterative analysis of the PDT. Intuitively, the PDT can be viewed as a kind of depth first search tree whose nodes are mapped to stages of the targeted pipeline. Any number of nodes of the PDT can be mapped to a single stage of the pipeline. PDT scheduling ultimately generates schedules with different latency/throughput trade-offs to effectively explore the multi-dimensional space of signal processing performance considerations. Furthermore, the PDT scheduling process can take into consideration various scheduling constraints, such as constraints on the number of available processors, and the amounts of on-chip and off-chip memory, as well as performance-related constraints (i.e., constraints involving latency and throughput).

The PDT scheduling approach places special emphasis on distinguishing and taking into account different modes of parallelism — task-level parallelism, as well as homogeneous and heterogeneous data parallelism — that must be exploited carefully to achieve efficient implementation of image processing applications. Data parallelism is a specialized form of parallel processing that allows multiple copies of a single task to execute simultaneously on multiple processing units. Heterogeneous data parallelism is an extension of data parallelism that allows for variability in the sizes of the memory regions to which data parallelism is applied. Under heterogeneous data parallelism, each copy of a task handles different sizes of blocks from the input data stream.

Although concepts related to the PDT and PDT scheduling can be applied to various domains of signal processing, including speech processing, high fidelity audio processing, and digital communications, the emphasis in PDT on data parallelism considerations makes the technique especially well suited to image processing.

Throughout the process of PDT scheduling, different interprocessor communication (IPC) architectures (point-to-point communication links or shared buses), and memory architectures (shared-memory or distributed memory architectures) are considered in an effort to achieve the most effective balance under the given constraints and available modes of parallelism.

2. RELATED WORK

In most practical contexts, scheduling applications onto multiprocessors environments is NP hard. Many deterministic heuris-

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

tics and evolutionary algorithm techniques have been proposed in this area (e.g., see [4][5]). In some cases, evolutionary algorithms are used in conjunction with deterministic approaches to yield their complementary advantages, and systematic methods have been developed also to perform such integration between evolutionary and deterministic approaches [2]. In particular, evolutionary approaches provide robust, easily adaptive methods for global search, while deterministic approaches are effective at exploiting application-specific insights that often provide for derivation of good solutions very rapidly, as well as effective local optimization. The PDT approach can be viewed as a deterministic approach that can be used in isolation as a fast, effective heuristic, and can also be combined with evolutionary algorithms when more thorough, computationally-intensive optimization is desired. This paper focuses on the former application of PDT scheduling; integration with evolutionary algorithms or other randomized search methods is a useful direction for further investigation.

A number of important deterministic techniques have been proposed in previous work related to embedded multiprocessor implementation of signal processing applications. Banerjee et al. [3] presented a two-step approach for coarse-grain pipeline scheduling by separating partitioning and process allocation for heterogeneous architectures. Hoang and Rabaey [6] developed a heuristic algorithm by innovative modeling and incorporation of interprocessor communication costs into the framework of coarse-grain pipelining. Konstantinides, Kaneshiro, and Tani [9] addressed detailed issues in modeling input/output (I/O) operations by decomposing I/O into sequential and parallel components.

PDT scheduling is different from these approaches in its deep integration of data parallelism configurations with task-level parallelism and coarse-grained pipeline implementation. Our PDT approach is motivated by the fundamental importance of data parallelism in performance optimization of image processing applications.

Incorporation of homogeneous data parallelism (parallelism across uniform-sized segments of data) and task duplication is reviewed in [10] in the context of non-pipelined schedules. In this paper, we develop novel techniques to model and exploit *heterogeneous* data parallelism, and to incorporate such an extended data parallelism formulation into the pipeline scheduling process.

Subhlok and Vondran [14] have previously considered the integration of data parallelism with task-level parallelism for multiprocessor scheduling. However, this work focuses mainly on applications that can be represented as linearly-chained dataflow graphs. Applying data parallelism and task parallelism to applications that have more general dataflow topologies causes various complications that are not addressed by the techniques of Subhlok.

In contrast, this paper targets general application dataflow topologies, including those with linear and non-linear data dependencies, and configures data parallelism and task parallelism appropriately based on the dataflow topology as well as the given implementation constraints. As with the techniques described above, we assume that for each task in the application dataflow graph, a reasonable estimate of the task's execution time is provided. Furthermore, as is common in the application of dataflow graphs to signal processing applications, we assume iterative execution of the dataflow graph, where the graph is executed repeatedly on an input data stream that is of indefinite (unbounded) length.

To demonstrate our proposed methods, we have applied them to complex morphological operations, Laplacian pyramid computation, Gaussian pyramid computation, and multi-resolution splines, which are all important image processing subsystems. The morphological operations that we have considered include opening, closing, gradient, Laplacian, smoothing and top-hat.

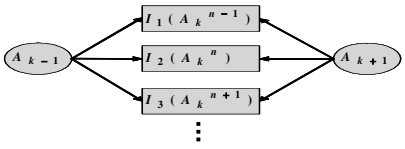
3. DATA PARALLELISM

3.1 Heterogeneous data parallelism

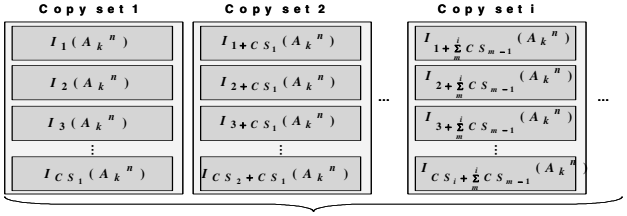
Data parallelism allows multiple copies of a single task to run on multiple processing units by task duplication when the operation of each task is independent. Each copied task processes a sub region of the whole data frame. The whole data frame can be divided into sub regions with different offsets. Finally the whole data frame is processed by each copied task in parallel. The sizes of data sub regions are identical for all copied tasks in conventional application of data parallelism.

Heterogeneous data parallelism is an extension of data parallelism that allows for dynamic change of the sub region size depending on the availability of resources, and the nature of the data parallel computations. In heterogeneous data parallelism, the whole data frame can be viewed as a collection of *copy sets*, where the size of the copy sets may or may not be the same. Each copy set consists of some number of sub regions that have identical sizes. Each copied task is allocated to handle a different copy set area. Inside each copy set, each task handles different sub regions. The size of sub regions within a copy set is the same, and can be obtained by dividing the size of the copy set by the number of task copies assigned to the corresponding copy set. The number of task copies within a copy set may vary from 1 to some number N, depending on the availability of processors. Each task copy corresponds to a separate invocation of the task, and in this way, different invocations of a single task processes different sub regions, and are allocated to different copy sets.

Figure 1 illustrates duplication of a task A_k under conventional (homogeneous) and heterogeneous data parallelism. In Figure 1a, each invocation of task A_k processes different sequential data frames. The first invocation of task A_k, which we denote by $I_1(A_k^{n-1})$, processes the $(n-1)$ th data frame, whereas $I_2(A_k^{n})$



a) Task duplication under conventional data parallelism.



b) Task duplication under heterogeneous data parallelism.

Figure 1. Task duplication under conventional data parallelism and under heterogeneous data parallelism.

and $I_3(A_k^{n+1})$ process the n th and $(n+1)$ th data frames, respectively. Therefore, as the number of invocations increases, the buffer size between A_{k-1} and A_{k+1} increases too.

In Figure 1b, the overall data frame is divided into several copy sets. Each copy set consists of varying-size sub regions, and is processed by different invocations of task A_k. Decisions on the number of invocations of a task in each copy set, the size of a copy set, and the sizes of sub regions within the copy sets are based on the availability of (idle) processors.

For a given task A_k, the following equations characterize the sizes of its copy sets:

$$SR_i = region(I_p(A_k^n)),\ p = j + \sum_{m=1}^{i} CS_{m-1}, \quad (1)$$

$$region(CS_i) = CS_i \times SR_i, \text{ and} \quad (2)$$

$$frame_n = \sum_{i=1}^{M} region(CS_i). \quad (3)$$

Here, SR_i characterizes the area processed by invocation p of task A_k within its i th copy set for n th data frame. This value is the same for all j within the associated copy set. $region(CS_i)$ represents the area processed within the i th copy set, and CS_i is the number of task invocations within the i th copy set ($CS_0 = 0$). M is the total number of copy sets for processing n th data frame, $frame_n$.

3.2 Clustering

Our heuristic scheduling algorithm for multiprocessors integrates both data parallelism — in heterogeneous form where appropriate — and task parallelism in a pipelined way based on user-specified constraints. To aid in this process, tasks that immediately follow a common predecessor task (in the application dataflow graph) are grouped into a single cluster for better utilization of shared memory architectures. This transformation step is skipped for distributed memory architectures.

After clustering, a new graph called the *cluster dependency graph,* which is acyclic is generated based on the dataflow relationships between clusters. The cluster dependency graph is used for partitioning clusters into stages of a pipeline. Each partition consists of a group of clusters. During partitioning, clusters are allocated to stages of the target pipeline. Clusters in each partition form a new cluster dependency graph within the corresponding partition. A cluster dependency graph satisfies a topological sort within each partition. We describe the partitioning process is in more detail in Section 4.2.

4. THE PIPELINE DECOMPOSITION TREE

In the macro-pipelined scheduling model that we target, the throughput is given by the reciprocal $(1/L_b)$, where L_b is the latency of the bottleneck stage in the synthesized pipeline. The overall latency, is given by $n_s L_b$, where n_s is the number of pipeline stages. Thus, in general, a derived pipeline configuration can be latency-optimal or throughput-optimal, or it can provide some trade-off between latency- and throughput-optimal performance based on the given resource constraints.

Ideally, the throughput can be improved by simply increasing the number of stages in the synthesized pipeline by sacrificing latency. However, an improperly decomposed pipeline degrades both throughput and latency. This paper provides a new representation called the **pipeline decomposition tree** (**PDT**) for exploring different macro-pipeline configurations and their associated latency/throughput trade-offs.

The process of PDT-based design space exploration can be viewed as a form of depth first search. Starting from the application dataflow graph, the PDT partitions the graph into two acyclic subgraphs. The objective is that clusters in each partition subgraph have relatively weak inter-cluster dependencies, so that each partition has more potential parallelism. Inter-cluster dependency becomes highest when all clusters in a subgraph are linearly linked (i.e., connected in a chain). On the other hand, cluster dependency is weakest when all clusters in a partition are independent.

Equations (4) and (5), which define expressions to minimize during the PDT construction process, are used to decompose a graph into two subgraphs π_1 and π_2 so that the cluster dependencies in each subgraph are evenly distributed, and the subgraphs and have similar overall execution times $executeTime(*)$:

$$min(\left|Dep(\pi_1) - Dep(\pi_2)\right|), \text{ and} \quad (4)$$

$$min(executeTime(\pi_1) - executeTime(\pi_2)). \quad (5)$$

Here, each cluster τ_i in each subgraph (π_1 or π_2) should satisfy:

$$if\ \tau_i \subset \pi_1,\ successors(\tau_i) \subset \pi_1 \text{ or } successors(\tau_i) \subset \pi_2, \text{ and}$$

$$if\ \tau_i \subset \pi_2,\ successors(\tau_i) \subset \pi_2.$$

Furthermore, the expressions used to assess cluster dependencies are given by

$$Dep(\pi) = e_{\pi_{cp}} + i \bullet e_{\pi_{\overline{cp}}}, \text{ and } |Dep(\pi)| = \sqrt{e_{\pi_{cp}}^2 + e_{\pi_{\overline{cp}}}^2}, \quad (6)$$

where

$$e_{\pi_{cp}} = \sum_{j=1}^{|\pi_{cp}|} j,\ e_{\pi_{\overline{cp}}} = \sum_{j=1}^{\left|CH_{\pi_{\overline{cp}}}\right|} \sum_{k=1}^{\left|CH_{j,\pi_{\overline{cp}}}\right|} k; \quad (7)$$

π_i represents the i th subgraph; $Dep(\pi)$ represents the degree of inter-cluster dependency in subgraph π; $e_{\pi_{cp}}$ represents the weighted sum (in terms of execution time) of critical path edges within subgraph π; $e_{\pi_{\overline{cp}}}$ represents the weighted sum of edges in non-critical paths within partition π; π_{cp} represents the number of critical path clusters within subgraph π; $\pi_{\overline{cp}}$ represents the number of clusters in non-critical paths within subgraph π.

Furthermore, the symbol CH represents a cluster chain, which is a maximal linear subgraph (i.e., no linear subgraph properly contains it) that contains two or more clusters within a partition. Each partition π can have one or more isolated CHs. The symbol $CH_{\pi_{\overline{cp}}}$ represents a set of CHs that reside along non-critical paths within partition π. Thus, $CH_{\pi_{\overline{cp}}}$ can be expressed as

$$CH_{\pi_{\overline{cp}}} = \left\{ CH_{1,\pi_{\overline{cp}}}, CH_{2,\pi_{\overline{cp}}}, \ldots, CH_{\left|CH_{\pi_{\overline{cp}}}\right|,\pi_{\overline{cp}}} \right\}, \quad (8)$$

where $\left|CH_{\pi_{\overline{cp}}}\right|$ is the number of CHs on non-critical paths, and $\left|CH_{k,\pi_{\overline{cp}}}\right|$ is the length of $CH_{k,\pi_{\overline{cp}}}$ in terms of cumulative execution time.

As stated above, $Dep(\pi)$ measures the degree of inter-clus-

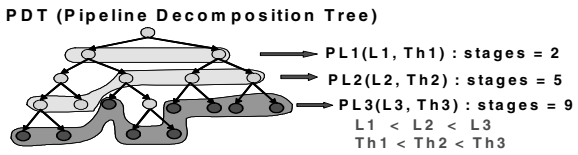


Figure 2. An example of pipelines generated by the PDT.

ter dependency within partition π. In this formulation, $Dep(\pi)$ is in general a complex number. The real part of $Dep(\pi)$ represents the weighted sum (in terms of task execution times) of critical path edges, while the imaginary part represents the weighted sum of edges in non-critical paths. The real part corresponds to a potential lower bound on the latency of the associated partition; this bound can be further reduced by exploiting heterogeneous data parallelism, as will be addressed later in the paper. The imaginary part characterizes the dependency degree of clusters on non-critical paths. Lower values for the components of $Dep(\pi)$ result in more parallelism during scheduling.

4.1 Mapping of PDT Nodes into Pipeline Structures

During the process of constructing the PDT, partitions are constructed recursively, and information about all partitions in the resulting hierarchy is stored. Partitions in intermediate levels provide for various pipeline organizations with various corresponding trade-offs between latency and throughput.

Intermediate-level PDT partitions can be mapped to different stages of various pipelines. The method we use for mapping partitions to stages is based on the distribution of $executeTime(\pi)$. Therefore, partitions at different depth levels of the PDT can be flexibly configured to generate various sets of pipelines. Figure 2 illustrates how pipelines can be constructed from PDT nodes at different depths. In this example, PL 1 has the best latency, PL3 provides the best throughput, and PL2 provides a trade-off between latency and throughput.

4.2 Critical Path Based Partitioning from the PDT

Based on the PDT representation developed above, this section introduces a novel heuristic technique, called *critical path based partitioning* (*CPAP*). CPAP divides a PDT-based partition into two sub-partitions in a manner that is driven by the formulations of Equations 4 and 5.

Our CPAP technique partitions clusters into sub-partitions by cutting critical paths of cluster dependency graphs evenly in terms of the estimated execution time of the clusters. With CPAP, the consideration of over- and under-loaded pipeline stages can be reduced before scheduling. CPAP is recursively applied to each level of the PDT for dividing partitions into sub-partitions until the application dataflow graph is fully scheduled onto the set of available target processors.

Figure 3 gives a more detailed outline of the CPAP algorithm., and Figure 4 illustrates how CPAP processes local critical path to divide parent partitions into sub-partitions.

5. INCORPORATING DATA PARALLELISM

In this section, we develop a technique called *heterogeneous data parallelism with earliest start times* (*HDEST*), to comprehensively take data parallelism considerations into account during the process of pipeline configuration. The HDEST algorithm is an extension of the widely-used EST (earliest start time) technique, which uses the

```
CPAP (G, CutTh) {
  CPAPDB = ⊥;
  LongestPath = ⊥;
  π_left = ⊥; π_right = ⊥; G_left = ⊥; G_right = ⊥;
  while (TRUE) {
    Step 1 => Find a critical path in the given graph.
    for(i=0;i<G.length;i++) {
      CP[i] = FindCP(G);
      if(LongestPath.length < CP[i].length)
        LongestPath = CP[i];
    }
    Step 2 => Add the left half of clusters of current longest
    path (LongestPath) to the left partition only if each node in
    LongestPath satisfies Equations 4 and 5.
    for((i=0;i<LongestPath.length;i++) {
      predeNodes[]=predecessor(LongestPath.node[i]);
      if(∀predeNodes = ⊥ || ∀predeNodes ⊂ π_left) {
        π_left += LongestPath.node[i];
        G_left.add(LongestPath.node[i]);
      }
      else
        break;
    }
    G = G - LongestPath;
    Step 3 => continue until executeTime(π_left) <= CutTh.
    if(executeTime(π_left) ≥ CutTh or G = ⊥)
      break;
  }
  π_right = G.nodes();
  G_right.add(G.nodes());
  CPAPDB = {π_left, π_right, G_left, G_right};
  return CPAPDB;
}
```

FindCP: return the critical path, *LongestPath* from graph, *G*.
predeNodes []: predecessors of a given node. *predecessor* (*n*); return predecessors of node, *n*. *CPAPDB*: a data base of partitions and graphs (π_{left}, π_{right} G_{left} and G_{right}) grouped by *CPAP*.

Figure 3. CPAP algorithm.

minimum starting time among available tasks as the main priority metric during scheduling [13].

HDEST applies a priority to a task in its scheduling ready list (*RL*) based on the depths of the critical paths associated with succeeding tasks. Thus, a task with the longest critical path of succeeding tasks has the highest priority in the *RL*.

HDEST also looks up all tasks in the *RL* and classifies them into two groups based on the existence or absence of heterogeneous data parallelism. In this classification, a task with heterogeneous data parallelism is called a *THD* task, and a task without heterogeneous data parallelism is called a *TNHD* task. In HDEST, idle processors are utilized by task duplication in conjunction with *THD*. Specifically, HDEST applies heterogeneous data parallelism for increasing the processor utilization (*PU*). The *PU* represents the fraction of time that a processor is active relative to the latency of a pipeline stage. In case every task in a stage is a *THD* task, all processors in the associated pipeline stage can be fully utilized.

In general, task dependencies and *TNHD*s prevent us from achieving an ideal *PU*. After HDEST, stages with poor *PU* are repartitioned by a refinement process that redistributes the workloads of different stages.

Figure 5 illustrates the HDEST process by showing how a pipeline configuration $C1$ derived by EST is improved to a new

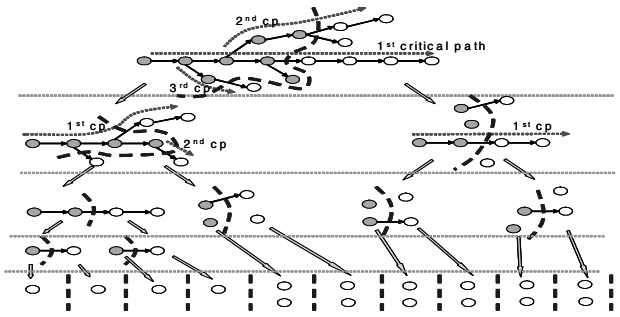


Figure 4. An example of the CPAP method.

configuration $C2$ by HDEST. This improvement arises from utilizing an idle interval of configuration $C1$ through application of heterogeneous data parallelism.

Because of the data parallelism available in THD tasks, such tasks can utilize, through task duplication, idle times of processors allocated to $TNHD$. If all tasks in a given stage are THD tasks, then all processors can be fully utilized by task duplication. On the other hand, if both THD and $THND$ tasks coexist in a given stage, idle times of processors caused by task dependency can be filled with THD tasks.

HDEST exploits heterogeneous data parallelism by using the earliest start time metric for candidate tasks. First, HDEST classifies all tasks in the scheduling ready list RL as THD and $TNHD$ tasks. Next, HDEST schedules the $TNHD$ tasks by considering their priorities and their communication costs, based on the targeted memory architecture. $TNHD$ tasks are scheduled before THD tasks because $TNHD$ tasks, due to their lower flexibility, are more critical in terms of data dependency. The overall execution times of THD tasks can be reduced further by exploiting heterogeneous data parallelism. When no tasks are available according to the ready list, task duplication is considered for THD tasks that have already been scheduled, and that overlap idle intervals. To idle processors that correspond to such idle intervals, HDEST applies task duplication to allocate copies of the associated THD tasks to the idle processors. Figure 6 outlines the HDEST algorithm in more detail.

6. EXPERIMENTAL RESULTS

This paper uses the Texas Instruments (TI c64x) Code Composer Studio to measure the estimated execution time of each task within the application dataflow graph as it runs on a single processor.

To evaluate our proposed pipeline configuration techniques, we applied the techniques to a suite of complex morphological applications, and to Laplacian pyramid, multi-resolution spline and MPEG2 encoder computation. Each application was scheduled under different constraints and architectures. For concreteness, we assumed that each DSP chip in the simulated target platform can integrate up to four processor cores, and that each DSP chip has on-chip memory and external memory. Each stage of a pipeline in the class of target architectures consists of one or more DSP chips with different numbers of processors cores. We assumed that external memory for each processor core within a DSP chip can be configured either as a separate-memory architecture (SP) or a shared-memory architecture (SH), whereas only a shared memory was allowed for on-chip memory due to chip-level size considerations.

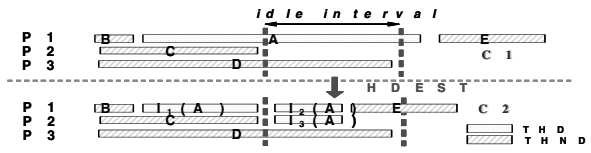


Figure 5. An example of a schedule derived by **HDEST**

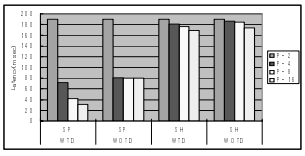
To explore the interaction of our proposed techniques with memory constraints, we applied 10% reduction for on-chip memory and 50% reduction for external memory compared to peak memory usage of each processor core, and we observed the effects of these memory constraints on performance in each architecture configuration. The resulting solutions are referred to as "constrained memory (C)" solutions, whereas the solutions that result when these memory constraints are not imposed are referred to as "unconstrained memory (NC)" solutions.

We compared the proposed techniques with the earliest start time (EST) algorithm, and we performed the experimentation with 2-, 4-, 8- and 16-processor systems.

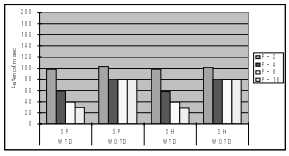
Figure 7 shows, for example, that scheduling results under memory constraints lead to 80% and 51% throughput degradation under a shared memory architecture with 16 processors, with and

HDEST() {
  Update the ready list *RL*;
  While (*tasks for scheduling* != empty) {
    if (*RL* != empty) {
    - *Select the task of highest priority;*
      → *TNHD tasks have higher priority than THD tasks*
      → *Tasks with more critical successors have higher priority*
    - *Choose the processor with the lowest communication cost;*
    - *Update RL;*
    }
    else {
      < *Exploit Heterogeneous Data Parallelism* >
      - *Check task duplication availability*
      - *Duplicate selected task based on available idle processors*
      - *Update RL*
    }
  }
}

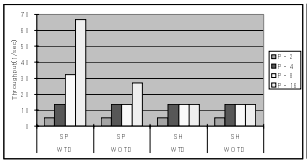
Figure 6. HDEST algorithm.



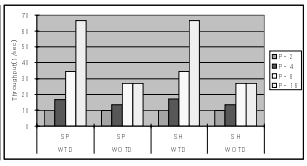
a) latency (constrained)



b) latency (unconstrained)



c) throughput (constrained)



d) throughput (unconstrained)

SP: separate memory. SH: shared memory. WTD: With Heterogeneous data parallelism. WOTD: Without Heterogeneous data parallelism. Constrained [SH: On-Chip 3.6KB, EX-MEM: the number of stages*64KB]. [SP: On-Chip 3.6KB, EX-MEM: the number of processors*64KB]. Unconstrained [SH: On-Chip 4KB, EX-MEM: the number of stages*2*64KB]. [SP: On-Chip 4KB, EX-MEM: the number of processors*2*64KB]

Figure 7. Latency and throughput comparison (multi-spline).

without heterogeneous data parallelism exploitation, respectively. A 16 processor, shared memory architecture can save up to 37.5% memory usage under unconstrained memory operation, while providing 25% better latency compared to a separate memory architecture. Applying heterogeneous data parallelism with our proposed scheduling techniques provides 2.46 times better throughput and 62.5% reduced latency compared to scheduling without heterogeneous data parallelism (WOTD), again for a 16 processor configuration.

Figure 8 shows a comparison between the EST and PDT technique for experiments with the multi-spline, Laplacian, image complex and MPEG2 applications. For example, the PDT technique provides 63.8% reduced latency and 4.94 times better throughput compared to the EST approach under an unconstrained memory configuration for the multi-spline application.

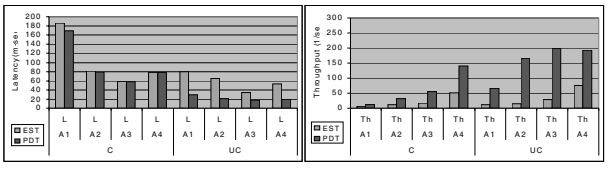
7. CONCLUSIONS

Effective, coarse-grained ("task-level" or "macro-") pipelined scheduling of an application over multiple processors generally provides increased throughput. However, pipelined scheduling can significantly increase latency. Furthermore, efficient pipelined scheduling of image processing applications requires careful and flexible integration of data- and task-level parallelism. This paper provides a new approach to generating coarse-grained pipelines for image processing applications in a manner that simultaneously considers latency/throughput trade-offs; memory and performance constraints; task-level parallelism; and homogeneous and heterogeneous modes of data parallelism. The approach is based on a novel data structure called the *pipeline decomposition tree* (PDT).

The PDT is useful for efficiently representing and exploring various sets of pipelining configurations that provide different trade-offs between latency and throughput. After pipelined schedules are generated through the PDT analysis process, a new technique called *heterogeneous data parallelism, with earliest start times* (HDEST) maps application tasks onto pipeline stages while considering memory and performance constraints. In the HDEST mapping process, heterogeneous data parallelism is carefully applied to improve both throughput and latency.

Our experimental results on various applications demonstrate the utility of the PDT data structure and HDEST mapping technique for embedded multiprocessor implementation of image processing applications. The applications in our experiments involved image processing because the emphasis in PDT on data parallelism considerations makes the technique especially well-suited for image processing.

The utility of PDT scheduling comes from the integrated exploitation of parallelism, along with its use of compile-time scheduling for predictable and low-overhead operation. The analysis process is performed at compile time while applying communication cost models depending on the underlying bus architecture and memory characteristics. Pipeline schedules derived by PDT scheduling can be mapped by hand into corresponding implementations in a straightforward manner. This mapping process can be automated through suitable code generation techniques, and development of such automaton is another useful direction for further work.



a) latency b) throughput

A1:Multi-Spline, A2:Laplacian, A3:ImageComplex, A4:MPEG2

C: Constrained memory. NC: Unconstrained memory

Figure 8. EST vs. PDT comparison,

8. REFERENCES

[1] S. Bakshi, D. Gajski, Partitioning and pipelining for performance-constrained hardware/software systems, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 4, p.419-432, Dec. 1999.

[2] N. K. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler. Systematic integration of parameterized local search in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 8(2):137-155, April 2004.

[3] S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman, Macropipelining based scheduling on high performance heterogeneous multiprocessor systems. *IEEE Trans. Signal Processing*, vol. 43, pp.1468-1484, June 1995.

[4] M. K. Dhodhi, I. Ahmad, and I. Ahmad. "A multiprocessor scheduling scheme using problem-space genetic algorithms". In *IEEE Conf. on Evolutionary Computation*, pages 214-219, 1994.

[5] D. E. Goldberg. "Genetic Algorithms in Search, Optimization and Machine Learning". Addison-Wesley, 1989.

[6] P. Hoang and J. Rabaey, Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. In *IEEE Transactions on Signal Processing*, vol. 41, no.6, June 1993.

[7] J. Jonsson, J. Vasell, "Real-Time Scheduling for Pipelined Execution of Data Flow Graphs on a Realistic Multiprocessor Architecture", In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, May 7-10, 1996, Atlanta, Georgia, USA, pp. 3314-3317.

[8] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, pages 291--307, February 1970.

[9] K. Konstantinides, R. T. Kaneshiro, and J. R. Tani, "Task Allocation and Scheduling Models for Multiprocessor Digital Signal Processing," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 38, no. 12, pp. 2151{2161, Dec. 1990.

[10] Y. K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, *ACM Computing Surveys*, vol. 31, no.4, p.406-471, Dec. 1999.

[11] S. Ranaweera, D. P. Agrawal, " A Task Duplication Based Scheduling Algorithm for Heterogeneous Systems", *International Parallel and Distributed Processing Symposium*, p 445-450, May 01 - 05, 2000, Cancun, Mexico.

[12] J. Rehg, K. Knobe, U. Ramachandran, R. S. Nikhil, Arun Chauhan, Integrated Task and Data Parallel Support for Dynamic Applications, *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, p.167-180, May 28-30, 1998.

[13] V. Sarkar. Partitioning and Scheduling Parallel Programs for Multiprocessor. The MIT Press, Cambridge, Massachusetts, 1989

[14] J. Subhlok, G. Vondran. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipeline. *SPAA, 1996, Padua, Italy.*