Inter-kernel Data Reuse and Pipelining on Chip-Multiprocessors for Multimedia Applications

Luis Angel D. Bathen, Yongjin Ahn, Nikil D. Dutt
Center for Embedded Computer Systems
University of California, Irvine, CA
{lbathen, yjahn, dutt}@uci.edu

Sudeep Pasricha
Department of Electrical and Computer Engineering
Colorado State University, Fort Collins, CO
sudeep@engr.colostate.edu

Abstract— **The increasing demand for low power and high performance multimedia embedded systems has motivated the need for effective solutions to satisfy application bandwidth and latency requirements under a tight power budget. As technology scales, it is imperative that applications are optimized to take full advantage of the underlying resources and meet both power and performance requirements. We propose a methodology capable of discovering and enabling parallelism opportunities via code transformations, efficiently distributing the computational load across resources, and minimizing unnecessary data transfers. Our approach decomposes the application's tasks into smaller units of computations called kernels, which are distributed and pipelined across the different processing resources. We exploit the ideas of *inter-kernel data reuse* to minimize unnecessary data transfers between kernels and *early execution edg*es to drive performance. Our experimental results on a JPEG2000 case study show up to 80% performance improvement and 60% dynamic power reduction over standard application mapping approaches.**

I. Introduction

The ever growing demand for media-rich embedded systems, limitations in the uniprocessor domain [1], and technology scaling have motivated the need for efficient low power platform solutions. Olukotun et al. [2] showed that chip multiprocessor (CMP) platforms perform 50-100% better than superscalar architectures for applications with high levels of parallelism. These platforms are well suited for emerging multimedia applications with high levels of parallelism such as JPEG2000 [3], and H.264 [4], where data partitioning allows for task level parallelism [5]. Today's embedded systems are deployed with heterogeneous on-chip memory hierarchies composed of small caches and/or software-controlled scratch-pad-memories (SPMs), where SPMs are favored over caches due to their increased predictability, and reduced area and power consumption [6]. In order to map an application efficiently onto multiprocessor platforms with limited resources, designers have to address both, the memory constraint problem as well as the load balancing problem. The application mapping process is composed of several steps that include the partitioning of the application into tasks, task scheduling and data placement for each of these tasks. Designers often look at task scheduling and data placement as two separate steps, where they try to optimize each step separately. This approach is not optimal because the choice of a schedule will impact data placement opportunities. Conversely, data placement affects how well a schedule performs. Therefore, when mapping an application on to a multiprocessor platform, designers must look at both scheduling and data mapping as one tightly coupled step.

This research project was funded in part by the Federal Cyber Service: Scholarship for Service (SFS) Program from the University of California, Irvine.

This paper proposes a methodology capable of discovering and enabling parallelism opportunities by partitioning tasks via code transformations, efficiently distributing the computational load across resources, and minimizing unnecessary data transfers. Our approach decomposes the application's tasks into smaller units of computations called kernels, which are distributed and pipelined across the different processing resources. We exploit the ideas of inter-kernel reuse to minimize unnecessary data transfers between kernels and early execution edges to drive performance. Our experimental results on a JPEG2000 case study show up to 80% performance improvement and 60% dynamic power reduction over standard application mapping approaches.

The rest of this paper is organized as follows. Section II describes this work's motivations and contributions. Section III describes the proposed methodology. Section IV discusses the assumptions made and the target architecture. Section V describes our pipelining heuristic. Section VI discusses our validation environment. Section VII covers related work. Section presents experimental results. Finally, section IX presents concluding remarks and future work.

II. Motivation and Contributions

A. Multimedia Applications: JPEG2000 Case Study

We target our approach to multimedia applications that have a streaming nature, which means that data enters at one point, and is then propagated through a series of filters (tasks). Most data reuse techniques [29-36] focus on intra-kernel reuse, whereas we focus on inter-kernel reuse. Due to the amount of data that multimedia applications process, it is very common to partition the data and process it independently.

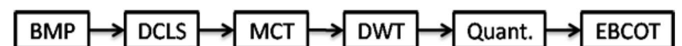

Figure 1. JPEG2000 Block Diagram.

The idea of inter-kernel data reuse can be observed by looking at the data flow in the JPEG2000 encoder from Figure 1. The image goes through a preprocessing phase that may include formatting the pixel value ranges (DCLS), multi-component transformation (MCT), and tiling (splitting

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.

the image into smaller images called tiles, which are processed independently). Next, each tile component is processed by the discrete wavelet transform (DWT) filter, which decomposes the image into multiple subbands at different resolution levels, where each subband (HH, HL, and LH) represents a down-sampled residual representation of the image, and the LL subband represents a 2:1 sub-sampled version of the original image component. This decomposition feature is one of the main motivations for the use of DWT as it facilitates the notion of progressive image transmission. The component array passes from containing component values to containing DWT coefficients. Next, DWT coefficients go through the scalar quantization step which improves the compression rate at the loss of quality. The array now contains the quantized values passed to the entropy coder block (EBCOT), which performs bit plane coding and arithmetic coding before generating the end bitstream.

```
void dcls
// input: B,G, R
// output: B,G, R
for ( i = 0; i < w; i++)
  for ( j = 0; j < h; j++) {
    B[i][j] = B[i][j]  - p(2, s-1)
    G[i][j] = B[i][j]  - p(2, s-1)
    R[i][j] = B[i][j]  - p(2, s-1)
  }
                    (a)
```

```
void mct
// input: B,G, R
// output: Yr, Ur, Vr
for ( i = 0; i < w; i++)
  for ( j = 0; j < h; j++)
    Yr[i][j] = ceil((R[i][j] +
      (2*(G[i][j] )) + B[i][j] )/4);
    Ur[i][j] = B[i][j] -  G[i][j];
    Vr[i][j] = R[i][j]  -  G[i][j];
                    (b)
```

Figure 2. Sample Code for the (a) DCLS and (b) MCT Tasks.

B. *Inter-kernel Data Reuse*

In standard task scheduling approaches, tasks are mapped without considering the notion of inter-kernel reuse. Each task might contain a series of computational kernels. Data for each task is mapped independently of whether the current task re-uses data from the previous task. Similarly, kernels within a task might not be considered when looking for reuse opportunities. Figure 2 shows two tasks, DCLS (a), and the multi-component transform MCT (b). From the sample code, we can see that MCT depends on the data [R, G, B arrays] produced by DCLS. Standard execution of MCT and DCLS would force MCT to execute after DCLS, meaning that DCLS would fill the cache and evict cache lines before MCT even has a chance to reuse them. In these types of tasks, the amount of intra-kernel reuse is limited, therefore, little benefit is gained from data-reuse techniques such as [29-34]. However, the amount of inter-kernel reuse is significant. By realizing that each of MCT's components will utilize at least two data streams produced by DCLS, we can reduce the number of unnecessary data transfers.

C. *Contributions*

In this paper we propose a methodology for the mapping of data intensive multimedia applications on multiprocessor platforms. The main contribution of this work is built on the idea of exploiting inter-kernel data reuse and load balancing to reduce power consumption due to unnecessary memory accesses and improve performance. The application goes through a series of code transformations which may include loop fission, loop tiling, and loop unrolling, as well as task/kernel level pipelining. The transformations enable us to exploit the application's parallelism and generate augmented task graphs for the applications, which are then pipelined with both power and performance as the cost functions.

III. PROPOSED METHODOLOGY

A. *Overview*

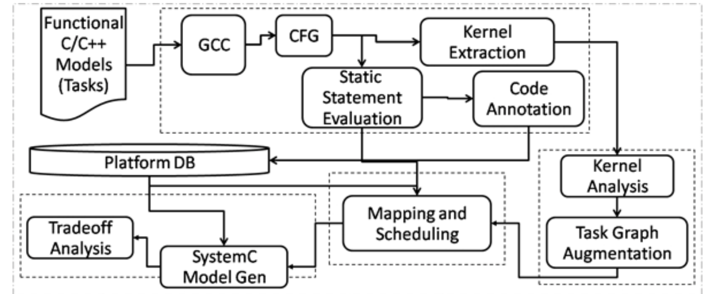

Figure 3. Methodology Overview.

The goal of our methodology is to exploit both parallelism as well as data-reuse between kernels in order to reduce the power consumption due to unnecessary data transfers between on-chip and off-chip memories. Figure 3 shows the block diagram of our methodology. The input to our methodology is an application already partitioned into a series of tasks in a functional model form. Each task represents a filter that operates over some input data stream. Once the task is done executing over that data stream, it is forwarded to the next task. Each can be composed of any number of computational kernels.

Our methodology is comprised of four main steps: *(i)* a front end, which is responsible for generating input models for our data/task mapping engine, and our simulation engine, *(ii)* a main analysis part for code transformation which performs the kernel analysis and task graph augmentation, *(iii)* an engine for data/task mapping and scheduling, and *(iv)* a component for the validation of our schedules/data placement by simulation. These steps are described in the following subsections.

B. *Input Model Generation*

The first step in our approach is to generate the necessary input models for our analysis. In order to accomplish this we need to extract the control flow graph (CFG) of the application from which we are able to extract the kernels for each task. Each basic block in the CFG is analyzed and the number of instructions executed as well as their types for each memory load/store is calculated. Each statement is statically annotated by examining its produced assembly code. This information is then used to annotate back the functional model with timing delays for the simulation model generation. Timing information is utilized by our analytical models during the scheduling and data mapping step.

C. *Task Graph Generation through Code Generation*

1) *Loop Fission*

Once we have identified the kernels for each task (by analyzing their CFGs), the next step is to decompose the task's execution into smaller units of computation in order to efficiently distribute the processing load across the available computational resources. We achieve this by subjecting each

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.

kernel to a series of loop transformations with the goal of exploiting as much parallelism as possible. Multiprocessor platforms benefit from loop fission because the loops can be distributed among the available processing cores. This process is similar to exploiting hidden parallelism in sequential applications [9].

(a)
```
void dcls:
for ( i = 0; i < w; i++)
  for ( j = 0; j < h; j++)
    B[i][j] = B[i][j] - p(2, s-1)
    G[i][j] = B[i][j] - p(2, s-1)
    R[i][j] = B[i][j]  - p(2, s-1)
```

(b)
```
void dcl_fission:
for ( i = 0; i < w; i++)
  for ( j = 0; j < h; j++)
    B[i][j]=B[i][j]- p(2, s-1)
for ( i = 0; i < w; i++)
  for ( j = 0; j < h; j++)
    G[i][j] = B[i][j] - p(2, s-1)
for ( i = 0; i < w; i++)
  for ( j = 0; j < h; j++)
    R[i][j] = B[i][j] - p(2, s-1)
```

Figure 4. DCLS (a) Before and (b) After Loop Fission.

Figure 4 shows the DCLS task before fission (a) and after loop fission (b). Since there are no intra-dependencies among the three arrays being accessed, it is possible to split the loop body into three independent loops (three independent kernels).

(a)
```
void dcls_tiled:
for( ii=0; ii<m; ii+=tw)
  for( jj=0; jj<n; jj+=th)
    for( i=ii; i<min(m, ii+tw); i++)
      for( j=jj+i; j<min(n+i, jj+th+i);
           j++)
        B[i][j-i] = B[i][j-i] - p(2, s-1)

for( ii=0; ii<m; ii+=tw)
  for( jj=0; jj<n; jj+=th)
    for( i=ii; i<min(m, ii+tw); i++)
      for( j=jj+i; j<min(n+i, jj+th+i);
           j++)
        G[i][j-i] = G[i][j-i] - p(2, s-1)

for( ii=0; ii<m; ii+=tw)
  for( jj=0; jj<n; jj+=th)
    for( i=ii; i<min(m, ii+tw); i++)
      for( j=jj+i; j<min(n+i, jj+th+i);
           j++)
        R[i][j-i] = R[i][j-i] - p(2, s-1)
```

(b)
```
void mct_tiled:
for(int ii=0; ii<m; ii+=tw)
  for(int jj=0; jj<n; jj+=tw)
    for(int i=ii; i<min(m, ii+tw); i++)
      for(int j=jj+i; j<min(n+i,
                 jj+tw+i); j++)
        Yr[i][j-i] = ceil((R[i][j-i] +
          (2*(G[i][j-i] )) + B[i][j-i] )/4);

for(int ii=0; ii<m; ii+=tw)
  for(int jj=0; jj<n; jj+=tw)
    for(int i=ii; i<min(m, ii+tw); i++)
      for(int j=jj+i; j<min(n+i,
                 jj+tw+i); j++)
        Ur[i][j-i] = B[i][j-i] - G[i][j-i];

for(int ii=0; ii<m; ii+=tw)
  for(int jj=0; jj<n; jj+=tw)
    for(int i=ii; i<min(m, ii+tw); i++)
      for(int j=jj+i; j<min(n+i,
                 jj+tw+i); j++)
        Vr[i][j-i] = R[i][j-i] - G[i][j-i];
```

Figure 5. Decomposition of DCLS (a) and MCT (b).

2) *Loop Tiling*

Looking back at Figure 2, we can see that the next task (MCT) depends on data produced by DCLS. Standard scheduling techniques would force MCT to wait until DCLS has completed its execution. Ideally, there would be enough cache/SPM space to store all three arrays, so by the time DCLS finishes executing, MCT can execute and hit the cache/SPM on every access to DCLS' produced data. However, embedded systems have limited memory resources, so storing an image in a 4KB cache/SPM is unrealistic. This brings us to loop tiling [10], which has been efficiently used to exploit loop level parallelism for both uniprocessors and multiprocessors [11]. The goal of loop tiling is to partition the iteration space into blocks, where each block accesses chunks of data elements, with the goal of increasing cache utilization. Classical approaches such as [11] try to distribute tile execution among the different computational resources for a *single* kernel. In our case, we try to pipeline the tile execution for a *series* of kernels in order to improve throughput as well as exploit inter-kernel reuse. The idea is that once we have decomposed the task into a series of computational kernels, we can further decompose them into even smaller computational units. Our tiling algorithm tries to optimize the tile size to fit into the available SPM space. Figure 5 shows the tiled kernel bodies of both DCLS and MCT tasks.

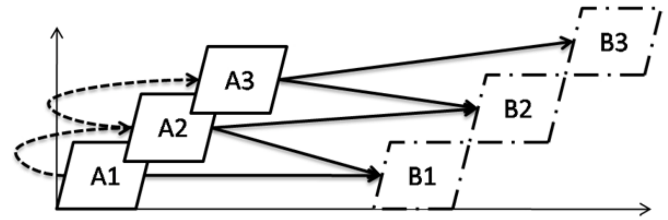

Figure 6. Inter (straight lines) and Intra (dashed lines) Dep. Edges.

3) *Tile Unrolling and Task Graph Generation*

Each tile can be represented as a single computational kernel with its own loop nest whose iteration space is bounded by the tile's boundaries. In Figure 5 we can observe the decomposition of DCLS and MCT into their respective computational kernels where each kernel has a series of data dependencies. So we have gone from two tasks (DCLS, MCT) and a single dependence to having six kernels which can be treated as six computational tasks themselves. Both DCLS and MCT can now be partitioned into three independent tasks, each operating over its own iteration space as well as its own data sets. The next step in our approach is to unroll the execution of the tiles generating a series of independent kernels, each operating over a block of data. During this step we create a task graph representing the unrolled kernels. Each node in the graph consists of a tile, and each edge represents tile dependence. There are two types of dependence edges generated during this step, inter-kernel and intra-kernel. Inter-kernel data dependencies refer to dependencies across kernels, such as the ones observed in Figure 5. Intra-kernel dependencies come from loop-carried dependencies, where the execution of the current tile depends on the execution of previous tiles. In order to resolve dependencies between kernels, we formulate Omega [12] tests for each tile pair based on their loop boundaries as well as their affine index expressions. Omega tests have been extensively used in the compiler community to resolve array dependencies and test for valid loop transformations. The goal of the omega test in our case is to find whether or not two tiles will have overlapping iteration spaces. If so, we can identify the bounds of the intersection between the two iteration spaces as well as the iteration in which such an overlap took place. Figure 6 shows the two types of edges generated during this step. Straight edges are inter-kernel dependencies, while the dashed edges are intra-kernel dependencies. Note that each node in the graph represents a single tile instance relative to its execution time. The Y-axis is the address of the array, and the X-axis is the relative iteration.

D. *Task Graph Augmentation*

1) *Augmenting the Existing Task Graph*

We now take these partial task graphs obtained from the previous step and augment our initial task graph, where the two

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.

nodes representing the DCLS and MCT tasks are decomposed into a series of smaller computational kernels where each kernel can operate over a smaller block of data. Figure 7 shows the gradual augmentation of the task graph for the DCLS and MCT tasks. Figure 7 (a) shows the initial task graph. Figure 7 (b) shows the initial task graph with both DCLS and MCT decomposed into their respective computational kernels (fission and tiling). Finally, Figure 7 (c) shows the augmented task graph with the execution of both DCLS and MCT further decomposed into multiple computational nodes, each with its own dependencies (after unrolling over the kernel tiles).

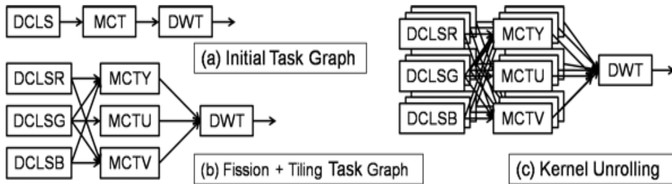

Figure 7. Progressive Augmentation of the Task Graph.

2) *Early Execution Edges*

Our work exploits the idea of early execution edges [13] which are a type of dependence edge that allow our scheduling heuristic to determine whether or not it is possible to start a task before its dependence finishes its entire execution. In order to obtain *early execution edges*, the live-range for a set of array accesses is calculated and utilized to find out at which iteration the given set is no longer referenced. Early execution edges are useful when a kernel cannot be tiled due to intra-kernel dependencies or tiling is not beneficial. An example of such task would be the discrete wavelet filter (DWT) which consists of a nested loop with three levels. The top level of the loop determines the level of decomposition while the lower two levels perform the low and high filters of the wavelet transform on the row and column pixels of the image being transformed. This process is repeated, each time over a reduce data set (1/4th the size of the previous step).

Figure 8. Wavelet Image Decomposition of Lena.

Figure 8 shows a two level decomposition of the image [14] by the DWT task. After each decomposition level is produced, DWT generates four independent subbands, three of which are produced by the kernel and ready to be processed by the next task, and the remaining subband (LL1 subband) goes through another round of decomposition. One observation that can be made during the execution of the task is that right after DWT has finished processing the first level of decomposition, the first level subbands (HH1, LH1, HL1) are available for the quantization task to start. Since quantization performs per-subband scalar quantization, it can process each subband independently. The quantization (Q) task is partitioned into two sub-tasks as shown in Figure 9, each operating over half or the original's iteration space. Say Q1 and Q2 are the two subtasks, where Q1 operates over the lower half of the image tile (subbands HL1, and the remaining lower level subbands), and Q2 operates over the upper half or the image tile (subbands HL1 and LH1). In this case, we observe that the iteration space between Q1 and DWT partially overlaps during the entire execution of the DWT task. Q2 on the other hand, operates over the upper half of the image, and its iteration space is composed of the top subbands of the first decomposition level. Thus, their iteration space will no longer overlap with DWT after the first level of decomposition. At this point, dependence edges between tasks Q1, Q2 and DWT, as well as their respective *early execution edges* are computed.

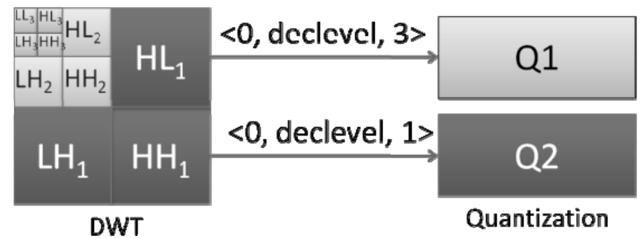

Figure 9. Early Execution Edges.

Early execution edges are calculated in a similar fashion as that of dependence edges. However, the goal of an early execution edge is to determine *when* the overlap between two kernels' iteration space will cease to exist. Once the iteration when such event is found, an early execution edge from task T_j to T_i containing the triplet <*loop level, iterator, iteration*> is created. Figure 9 shows three levels of decomposition of the DWT task, its respective subbands as well as the early execution edges for Q2 and Q1 with the loop level, iterator and iteration number triplets

3) *Clustering*

Once the final task graph has been computed, the next step is to determine whether or not clustering tasks will improve performance and reduce power. Clustering is needed because our approach increases not only the number of computational nodes in the task graph but also the number of dependencies that need to be resolved before a task is executed.

Assume a 32CPU CMP with 4KB SPMs. The original task graph shown in Figure 7 (a) would not be taking full advantage of the computational resources available because it would utilize at most 5 processors with pipelining. Assume the task graph is augmented to process 128x128 tiles independently. This means that the augmented task graph would execute DCLS and MCT sequentially, and then execute the remaining filters 3*64 times (1024x1024 image, 128x128 tile, 3 components). Such a task graph would take advantage of the computational resources. However, because 128x128 size image tiles cannot fit into a 4KB SPM, neither power nor performance would be optimal due to excessive misses. If we tile DCLS/MCT as well as Q/EBCOT in a way such that each kernel computes 4KB of data, we obtain an augmented task graph such as the one in Figure 10. In this case, a single instance (image tile) in the task graph will have a total of 30 tasks and the process would have to be repeated 64 times. So the number of task has gone from the original 5 to 30. The augmented task graph allows us to exploit

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.

the platform's computational and memory resources. If we decided to fully unroll the execution of the kernels from one instance to *n (n-degree unrolling)*, then we will have 30*n tasks to schedule. The number of tasks in the task graph depends on the SPM size (which drives the tile size selection) and the amount of parallelism we wish to exploit. At some point the amount of tasks generated will be too large, and instead of improving performance, we might end up with performance degradation due to the large amount of data dependencies and increased power consumption due to the thrashing effect.

We consider two types of clustering. The first is called *same type clustering (STC)*, where we decide to cluster each computational node into groups of tiles instead of having single-unit tiles as the base of our computational nodes. This will help us reduce the number of dependencies, the time it would take to schedule the tasks and reduce the thrashing effect since less tasks will be competing for resources. The second type is called *communication elimination clustering (CEC),* which tries to minimize unnecessary data transfers between tasks, maximizing inter-kernel reuse. This is accomplished by assigning the tasks that operate over the same data sets to the same cluster.

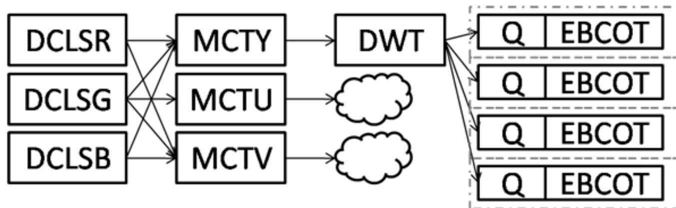

Figure 10. Communication Elimination Clustering of the Task Graph (Super Node: Quantization and EBCOT).

Figure 10 shows one instance (tile) of the JPEG2000 augmented task graph, with the quantization and the entropy coding tasks clustered to force their reuse of the quantized subbands. The clouds represent the remaining tasks. This technique is similar to the edge zeroing clustering approach introduced in [15]. The main difference is that we are working with smaller data sets, and the clustering is done on the basis of data-reuse between tasks. In some cases, *CEC* may cluster too many tasks into the same node, which may lead to performance degradation. This happens because it might be the case that clustering keeps our scheduler from taking advantage of available resources and existing early execution edges. Similarly, because our tiling is done with the memory constraint in mind, *STC* might lead to unnecessary data misses limiting the inter-kernel reuse.

4) Code Size Impact

Managing the data loaded and unloaded from the SPM as well as code transformations such as loop fission, loop tiling, unrolling, etc. are known to have an impact in both the code size. Our task decomposition approach replicates the code in the application's kernels obtaining similar code size increments ranging from 1.6 to 3.2 times in assembly lines for each kernel as observed in [33].

IV. Target Platform and Assumptions

A. Target Platform

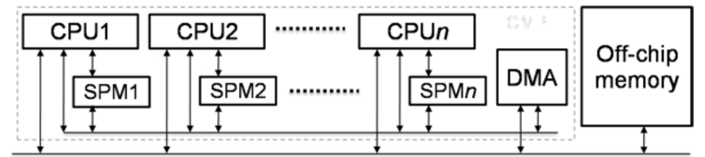

Figure 11. CMP Architecture.

Figure 11 shows our architectural model, which is similar to the one used in [7]. We consider a simple homogeneous CMP architecture, consisting of multiple processing cores, each with its own SPM, instruction cache, and a DMA engine to facilitate the data transfers among the various SPMs. We make use of a shared bus communication infrastructure since they are still the most dominant types of communication fabrics used in embedded systems.. Each CPU can talk to the DMA engine, and request transfers to/from main memory to any of the SPMs in the system as well as SPM to SPM transfers. Each CPU can talk to each of the SPMs through the SPM shared bus, at the cost of some extra communication cycles. Each CPU can also talk to off-chip memory through the off-chip shared bus.

B. Assumptions

Our approach assumes that the designer has already partitioned the application into a series of tasks (nodes) and their dependences (edges). Each application's task graph is known before hand, so their initial dependencies are known. Our transformation techniques work on applications with high levels of regular access patterns. We assume that each task's kernels can be represented as a series of well defined loop nests, which contain array accesses with affine loop expressions. Issenin et al. [8] proposed a method to automatically generate affine functions for loop nests from original C programs. Tasks with high levels or irregularity are still part of our task graphs, but are omitted from the code transformation path.

V. Mapping and Scheduling

A. Mapping Algorithm

In this paper, we use the quantum-inspired evolutionary algorithm (QEA) [16] to map tasks onto multiprocessors [17]. QEA has the advantage of obtaining a set of best alternative solutions at the same time under multiple criteria. Its efficiency has been proven by comparing it to similar evolutionary algorithms as well as exhaustive formulations [18, 19]. We briefly go over QEA. Refer to [16] for more details.

Figure 12 shows the overall mapping flow of our approach. First, given an input model, QEA generates a series of mapping solutions. Like other evolutionary algorithms, QEA is also characterized by the representation of the individual state, the evaluation function, and the population dynamics. The main difference is that it uses a probabilistic representation called Q-bit instead of binary, numeric, or symbolic representation. A Q-bit is the smallest unit of information (e.g. information on task mapping to CPU0 or CPU1) which is defined with a pair of numbers (α, β) where $|\alpha|^2 + |\beta|^2 = 1$. $|\alpha|^2$ gives the probability of

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.

the Q-bit to be found in the '0' state and $|\beta|^2$ gives the probability of the Q-bit to be found in the '1' state. A Q-bit may be in the '1' state, in the '0' state, or in a linear superposition of the two and this is how QEA keeps potential solutions in a compact way, thereby enabling much faster design space exploration than any other currently known evolutionary algorithm.

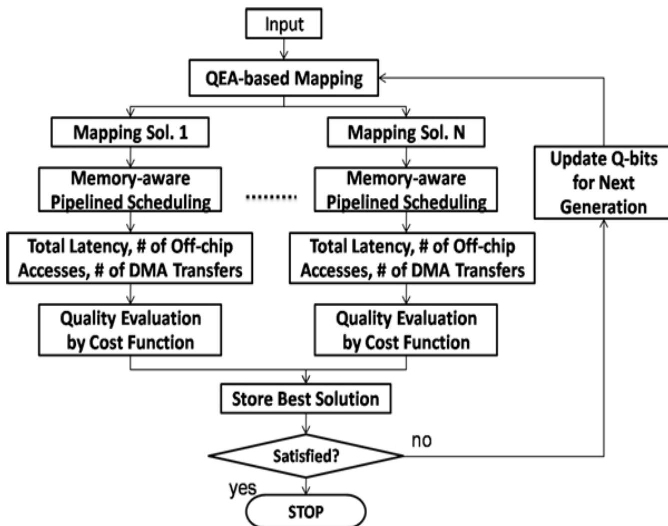

Figure 12. Overall Mapping and Scheduling Flow.

All tasks are represented by Q-bits, which are initialized to have the same probability ($\alpha, \beta = 1/\sqrt{2}$). Next, the mapping algorithm randomly generates a set of solutions by comparing the probability $|\beta|^2$ of each Q-bit with a random variable whose range is from '0' to '1'. For example, if $|\beta|^2$ is bigger than the random variable, the Q-bit can be mapped to CPU1 processor and if not it is mapped to CPU0.

The scheduling algorithm outputs the total latency, the number of off-chip accesses, and the number of DMA transfers for each mapping solution. With these outputs, the mapping algorithm evaluates the quality of the solution by using the cost function specified as a weighted sum. The function drives solutions to have as few off-chip accesses as possible and as less DMA access and latency as possible at the same time.

Finally, the solution with the best gain is used to update the probability of the Q-bits for next generations. If the state of a Q-bit in the best solution is '1', the algorithm drives the probability of the Q-bit of the offspring towards '1'. Here, $|\beta|^2$ (the prob. of "1" state) is increased thus the possibility to generate a better solution in the next generation increases.

B. Memory Aware Pipelined Scheduling

Once a mapping solution is found during the QEA step, the solution is evaluated by our memory aware pipelined scheduler. Our memory-aware pipelined scheduling heuristic is based on the idea of retiming [20]. Retiming relies on the notion of the Minimum Initiation Interval (MII), which serves as the minimum window for the execution of a set of tasks. The smaller the MII the higher the throughput is. MII is calculated by analyzing two types of MIIs, Resource constrained MII (ResMII) and Recurrence constrained MII (RecMII) [21]. ResMII is the maximum among ResIIs of all resources and RecMII is the maximum among RecIIs for all inter-iteration dependencies. ResII is the sum of execution times of all tasks mapped to a resource and RecII is the time elapsed by an inter-iteration dependency. MII is set to the maximum between ResMII and RecMII..

Next, list scheduling is used to schedule all tasks within the calculated MII considering communication costs including communication delay and bus conflict. If the mapping is not schedulable within MII, retiming is done for all possible tasks by converting an ILD (Intra Loop Dependency) into an LCD (Loop Carried Dependency) which does not constrain the scheduler. Thus, the retimed tasks are migrated to a different pipeline stage. In case the current mapping is still not schedulable with the current MII, the MII is incremented by a determined constant factor or one percent of the current MII.

Our list scheduler sorts all tasks in descending order based on their original execution time obtained through profiling. Tasks with lower execution time are more flexible to when scheduling them as their impact on the critical path is not as great as tasks with long execution times. When a given task can start depends on whether its dependences have finished their execution or if there are early execution edges for the task. For every early execution edge going to the task, we use the edge's information to determine if it is possible to execute the task at the current time without having to wait for its dependence to complete its execution. We verify that all dependencies meet the early execution criteria by looking at their current loop's iterator/iteration pair. If these match the iterator/iteration pair in the early execution edge, we can assume that we can start the execution of the task. The mappings are already given by the mapping algorithm as explained in Section V.A. The goal of our memory aware scheduling heuristic is to determine the mapping between each task's data and the memory space in order to minimize off-chip memory accesses and DMA transfers.

Before we schedule a task (*t*) mapped to CPU (*p*), we look at the data size currently placed in *p*'s SPM. If there is currently enough space for *t*'s data to be placed in the SPM, the data is mapped to the SPM. Otherwise, we next search for the remote available SPM which has enough space. Our approach tries to map the data to on-chip SPM as much as possible to minimize the number of off-chip accesses. If the data of one task which is already scheduled is currently mapped to a remote SPM or off-chip memory, another task depending on the data can bring the data to local SPM by using DMA only when the local SPM has enough space. Note that this step is critical as what data is placed on the SPM will affect the execution of the current schedule. So each potential data mapping will have an effect on the schedule's execution time.

VI. SOLUTION VALIDATION

Once the best schedule/data mapping solution has been found, the next step is to validate it, and obtain its dynamic power and performance through our simulation workbench [40, 41]. Our performance model generation is similar to the one proposed by [22]. During the input generation process we extract basic block information from the generated CFG for each task. This information is used to statically annotate the

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.

functional model with timing delays modeled as wait statements in SystemC. Once these delays have been inserted into the code we automatically generate our performance models. Because we are mainly interested in reducing the amount of unnecessary data transfers to off-chip memory, our goal is to capture the memory transactions on the bus and across the different memories as accurately as possible. Our memories and our bus are cycle accurate at the transactional level. Annotating the task with timing delays for each statement being executed allows us to capture the basic functionality and delay information to quickly evaluate the produced schedule. This is especially important given that we are targeting multiprocessor platforms. We capture both bus switching activity and memory transactions in order to evaluate the impact our transformations will have on both the dynamic power consumption and performance of the system.

VII. Related Work

A. Task Scheduling and Pipelining

The task scheduling problem has been investigated extensively in the field of parallel computing and has been proven to be NP-complete [23]. Typically, the scheduling algorithms start with a precedence graph and attempt to optimally map tasks to a series of processors. Most of these algorithms [24, 25] are greedy heuristics that rely on list scheduling. The amount of parallelism exploited by these techniques is limited, so researchers have continuously searched for new methods to parallelize the execution of their applications. One efficient way towards performance maximization is pipelining. Banerjee et al. [23] have proposed the macro-pipelining scheduling method for heterogeneous systems and compared conventional homogeneous and heterogeneous multiprocessor systems. Chata et al. [20] have developed a retiming heuristic called RECOD to optimize resource and memory utilization by partitioning the system. Bakshi et al. [24] have proposed a system level design method for pipelined implementation in hardware/software codesign. Shee et al. [25, 26] proposed pipelining a system with multiple heterogeneous Application Specific Instruction-set Processors (ASIPs) using a heuristic to explore the design space of such systems and obtain optimal configurations. Ko et al. [27] have extended the concept of data parallelism to heterogeneous data parallelism and proposed pipeline decomposition trees (PDTs) to explore various pipeline configurations. Gordon et al. [28] proposed a stream compiler to exploit task, data, and pipeline parallelism. None of these approaches takes into account memory and power as cost functions like our approach does.

B. Data Placement

Unlike cache-based platforms where data is dynamically loaded into the cache with hopes of some degree of reuse due to access locality, SPM based systems depend completely on the compiler to determine what data to load. Placement of data onto memory is often done statically by the compiler through static analysis or application profiling, the location of data is known prior to runtime which increases the predictability of the system. Panda et al. [29] profiled the application and tried to allocate all scalar variables onto the SPMs by identifying candidate arrays for placement onto the SPMs based on the number of accesses to the arrays, and their sizes. Verma et al. [30] look at an application's arrays, and identify candidate arrays for splitting. The goal is to find an optimal split point in order to map the most commonly used area of the array to SPM, leaving remaining array elements in main memory. Brockmeyer et al. [31], look at arrays, or parts of arrays, and determine copy candidates (CCs) based on the array's reuse information, once they identify a CC they try to optimize the assignment of the CCs to each level in the memory hierarchy. Kandemir et al. [32] use loop transformation techniques such as tiling to improve data locality in loop nests with array accesses, and map array sections to different levels in the memory hierarchy. Issenin et al. [33, 34] proposed a data reuse analysis technique for uniprocessor and multiprocessor systems that statically analyses the affine index expressions of arrays in loop nests in order to find data reuse patterns. They derive buffer sizes to hold these reused data sets, and could be implemented on the available SPMs in the memory hierarchy. Cho et al. [35] proposed a data allocation and clustering algorithm for multiprocessor systems, where they group data accesses for both regular and irregular arrays into clusters which are mapped onto the SPMs. Our approach differs from these techniques in that most of these techniques try to exploit intra-kernel reuse, where as we try to exploit inter-kernel reuse. Another difference in our approach is none of these techniques consider different schedules for their data placements.

C. Memory Aware Heuristics

Suhendra et al. [7] propose an ILP formulation that combines task scheduling, SPM partitioning, and data allocation. They show that by combining task scheduling and data placement into a single problem, they can achieve high gains in performance. Szymanek et al. [36] proposed a constructive memory aware scheduling algorithm that builds a schedule around the critical path and progressively schedules tasks in order to balance the memory utilization across processors. The main difference in our work from existing approaches is that we are exploiting the application's parallelism, pipelining, and data-reuse opportunities by applying different source level transformations to the application's tasks. These transformations allow us to break the workload into smaller units of computation, where each kernel has the potential of reusing data from a previous kernel. We distribute computations among the different computational resources with the ultimate goal of reducing unnecessary data transfers.

VIII. Experimental Results

A. Experimental Setup

Our performance models are built on top of the SystemC 2.0 [37] library. To evaluate the dynamic power due to memory accesses we used the CACTI v5.2 [38] tool set assuming 65nm SRAM technology. We used the Simplescalar 3.0 tool set [39] to generate assembly code (RISC ISA) for each task. To calculate execution time for each task we used simplesim from Simplescalar.

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.

B. The Base Cases

1) Basic Parallelized Task Graph

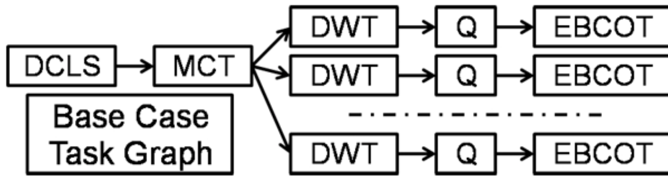

Figure 13. Base Case Task.

Our experimental setup consists of two base cases. The first base case is the classic CMP with SPMs of the same size (*Base SPM)*. In this approach the most commonly used data for each task will be mapped onto the SPM space, taking advantage of DMA burst features. The application is first profiled and the most commonly used data is loaded via DMA at run time. The second base case is the classical CMP with local caches (*Base Cache*) of the same size as the *Base SPM*, configured as two-way set associative caches, with 64 Byte cache lines, LRU policy and write through enabled. The base task graph parallelizes the execution of the DWT, Quantization and EBCOT tasks over the available *n* processors (for a CMP consisting of *n* processors). Both DCLS and MCT are executing sequentially, and then all image tiles (128x128 pixels each) are parallelized on the *n* processors. Figure 13 shows an example of the original task graph, which is mapped onto the two types of CMP platforms. Note that neither base case takes into account scheduling during the allocation process.

2) Our Augmented Task Graph

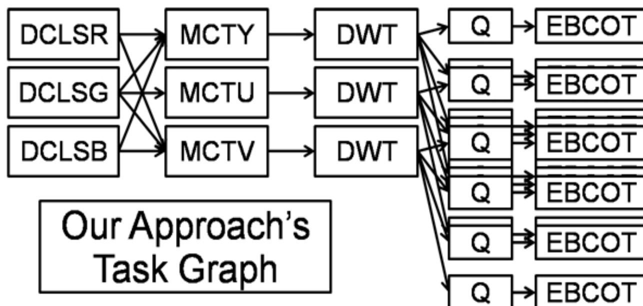

Figure 14. Augmented Task Graph for a Single Instance.

Figure 14 shows the case where we optimized each task except for DWT to operate over 4KB chunks of data. In this case, we pipeline the task on a CMP with SPMs (*Pipelined SPM*). Each of the different CMP configurations will have a different size augmented task graph since our code transformations are optimized to take full advantage of the available resources. The last of the test cases is the memory aware pipelined and clustering approach (*Pipelined SPM+Clustering*). The clustering approach forces inter-kernel reuse during the task graph generation step. Clustering helps reduce both, unnecessary memory transfers, which might be overlooked during the memory aware pipelining step as well as the number of tasks and their dependencies, speeding up the scheduling process.

C. Performance Comparison

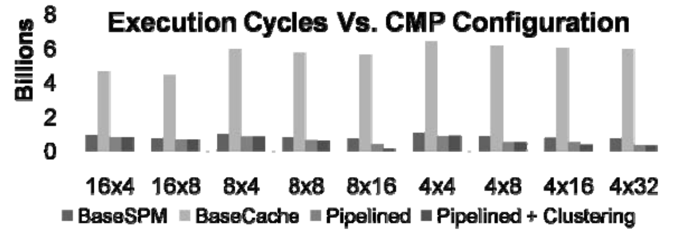

Figure 15. Execution Cycles vs. CMP Configuration.

Figure 15 shows the end-to-end execution cycles of the JPEG2000 encoder mapped onto a CMP configuration with varying CPU and memory sizes. As can be seen from the figure, all of the SPM approaches outperform the *Base Cache* approach. This is because we are dealing with a streaming application where the amount of intra-kernel reuse is limited, so the amount of evictions due to misses is extremely large. If we choose a smarter approach where we map the most commonly used data onto SPM statically such as the case of the *Base SPM* case, we can potentially see better performance improvements.

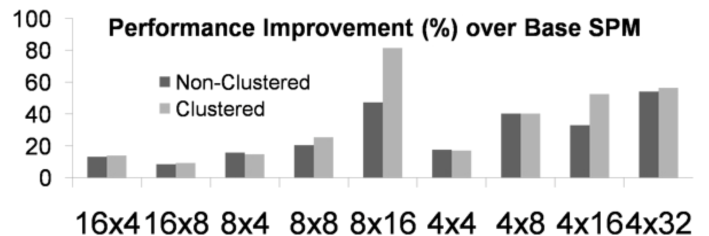

Figure 16. Performance Improvement over the Base Case.

Figure 16 shows the percentage of performance improvements over the Base SPM case for both, our memory aware pipelined approach, as well as our memory aware pipelined approach with clustering. As we can see in the results, the clustering approach performs better than the non-clustered approach in most cases because we reduce the number of unnecessary memory transfers by mapping the kernels that operate over the same data onto the same CMP, and reuse the same resources the previous tasks were using. In some cases, this is not the case, as we can see in the 8KB/4CPU case where pipelining the non-clustered task graph actually works best. This happens because there are cases when computational resources are idle and could be utilized by ready tasks, but because tasks were clustered, we would not be able to take full advantage of these opportunities.

D. Power Comparison

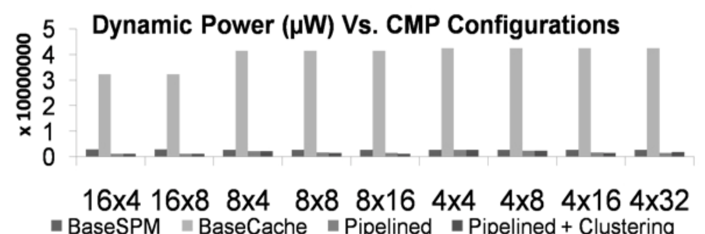

Figure 17. Dynamic Power (µW) vs. CMP Configuration.

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.

Figure 17 shows the dynamic power consumption for all four cases. As we can see, the SPM cases outperform the cache approach not only because of the fact that the data allocation is done efficiently for each of the SPM cases, but also because SPMs lack the hardware support caches have to dynamically load/offload data onto the lower levels of the memory hierarchy. All the added extra hardware support such as the tag-RAM, support of the LRU policy, etc. consumes extra power since each tag has to be checked during every read/write to the cache.

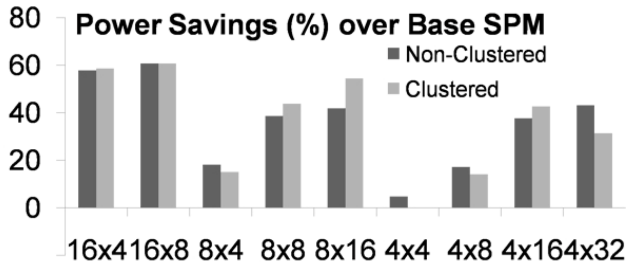

Figure 18. Power Savings over the Base Case.

Figure 18 shows the power savings over the *Base SPM* case. As we can see, the benefits of clustering and non-clustering vary from case to case. The reason is that in some cases, if we cluster more than one task into a computational node, this means that the new node might consume more resources than the non-clustered tasks leading to the extra misses to main memory.

E. Pipelining Multiple Instances

In section III.D we described the idea of task graph generation and augmentation. The size of the task graph is directly proportional to the degree of unrolling done during the tiling process. Figure 14 showed a single instance of the task graph generated by our approach with unrolling degree of 1 (33 tasks to pipeline). If we increased the degree of unrolling to 2, we double the number of pipelined tasks. Figure 19 shows the performance obtained by having different degrees of unrolling on different memory sizes and number of processors. The closer a data point is to the center the better its performance is.

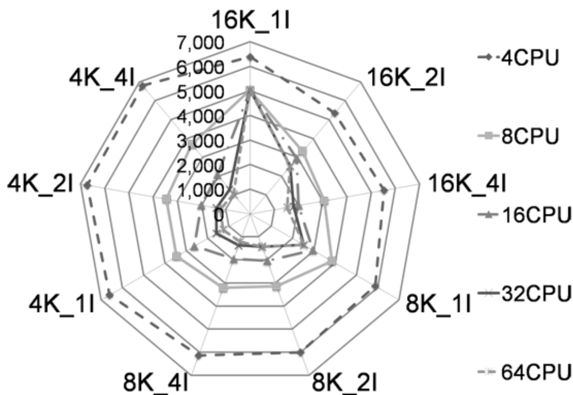

Figure 19. Exploration Space for Multiple Instance Pipelining. Execution Time Scale is in Million Cycles.

The 16K_1I case which corresponds to a CMP of varying number of CPUs, 16KB of SPM size and degree of 1 unrolling performs worst in general than both the 16KB SPM, 2 and 4 degrees of unrolling (16K_2I and 16K_4I). This leads us to think that as we increase the degree of unrolling, performance will increase. This is not always the case as we can see in the 4KB SPM case, where pipelining 4 instances (unrolling degree of 4) performs worst than the single instance (unrolling degree of 1), for both the 4CPU and 8CPU CMPs. This motivates the need to explore what the optimal degree of unrolling would be for the target platform. To help us find the most ideal unrolling degree, we can formulate the problem as a software pipelining problem where each task in the task graph can be considered an instruction. Each instruction's delay is simply the task's execution time.

F. Algorithm Runtime

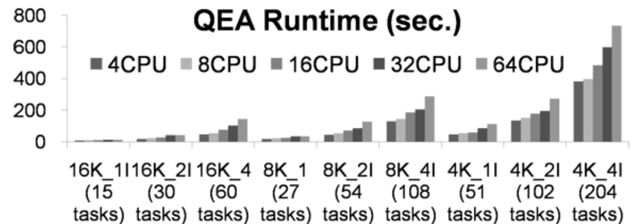

Figure 20. QEA Runtime (seconds) as a Function of the Degree of Unrolling and Memory Size.

Figure 20 shows the runtime of the QEA based memory aware scheduler as a function of the degree of unrolling, which is directly proportional to the number of tasks the QEA scheduler will try to pipeline on the given platform. We used same number of generation and population size in the QEA for all the cases (generation=3000, population=25). As we increase unrolling degree (number of tasks), and the number of CPUs, the time it takes QEA to pipeline the task graph goes from 8 seconds for the 4CPU/16KB CMP and unrolling degree of 1 to about 12 minutes for the 64CPU/4KB CMP and unrolling degree of 4. This shows just a 90x runtime increase in QEA while the search space increases about 870x. Note that this is because we use the same generation number and population size just for consistent comparison for all the cases. In fact, in the case of smaller number of CPUs and tasks (e.g. 4CPU/15tasks), 100 and 5 for the generation number and the population number respectively are sufficient to find a good solution. There are two steps that can be taken to help minimize the QEA runtime. The first is to limit the degree of unrolling performed, thereby reducing tasks and dependencies. The second step is to perform aggressively cluster more tasks before the graph generation step, thereby reducing the number of tasks that need to be considered by our scheduler.

IX. CONCLUSION AND FUTURE WORK

In this paper we proposed a software mapping and scheduling methodology for multimedia and data intensive applications. Our methodology targets chip-multiprocessors with scratchpad memory support. Our methodology progressively transforms the application's code in order to discover both inter-kernel data reuse as well as parallelism opportunities. Once these opportunities have been discovered, our memory aware scheduler pipelines the execution of the tasks onto the target platform. By doing this we are increasing the application's

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.

throughput as well as maximizing inter-kernel reuse, efficiently minimizing unnecessary off-chip data transfers. Our experiments on the JPEG2000 case study show an average of 31% performance improvements and 35% in power reduction when compared to the base case. In some cases we saw up to 80% performance improvements and 60% in power reduction. We are currently exploring the impact of our technique on applications with greater levels of intra-kernel reuse. We are also interested in exploring more complex platforms such as heterogeneous MPSoCs, with more complex memory hierarchies as well as different types of communication infrastructures such as point-to-point, bus matrix, and network-on-chips.

REFERENCES

[1] M. S. Hrishikesh et al., "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," ISCA, pp. 248, 2000.

[2] K. Olukotun et al., "The Case for a Single-chip Multiprocessor," SIGPLAN, pp. 2-11, 1996.

[3] JPEG2000, ISO standard, ISO/IEC 15444-1, 2000.

[4] ITU-T Rec. H.264, ISO/IEC 14496-10, 2003.

[5] J. Chong et al., "Efficient Parallelization of H.264 Decoding with Macro Block Level Scheduling," ICME, pp. 1874-1877, 2007.

[6] R. Banakar et al., "Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems," CODES, pp. 73-78, 2002.

[7] V. Suhendra et al., "Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures," CASES, pp. 401-410, 2006.

[8] I. Issenin et al., "FORAY-GEN: Automatic Generation of Affine Functions for Memory Optimizations," DATE, pp. 808-813, 2005.

[9] H. Zhong et al., "Uncovering Hidden Loop Level Parallelism in Sequential Applications," HPCA, pp. 290-301, 2008.

[10] M. Wolfe, "More Iteration Space Tiling," Supercomputing'89, pp. 655-664, 1989

[11] J. Xue, "Loop Tiling for Parallelism," Kluwer Academic Publishers, 2000.

[12] W. Pugh, "The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis," ACM/IEEE Conf. on Supercomputing, pp. 4-13, 1991.

[13] L. Bathen et al., "A Framework for Memory-aware Multimedia Application Mapping on Chip-Multiprocessors," ESTImedia, pp. 89-94, 2008.

[14] Lena: http://www.cs.uiowa.edu/~jorgen/Haar.html.

[15] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors," Cambridge, MIT Press. 1989.

[16] K. Han et al., "Quantum-inspired evolutionary algorithm for a class of combinatorial optimization," IEEE TEC 6, pp.580-583, 2002.

[17] Y. Ahn et al., "SoCDAL: System-on-chip Design Accelerator," ACM TODAES 17, pp. 1-38, 2008.

[18] S. Kirkpatrick et al., "Optimization by Simulated Annealing," Science 220, pp. 671-680, 1983.

[19] Yang, H. et al., "Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC," DATE, pp. 69-74, 2009.

[20] K. Chatha et al., "RECOD: A retiming heuristic to optimize resource and memory utilization in HW/SW codesign". CODES, pp. 139-144, 1998.

[21] B. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," Int. Symposium on Microarchitecture, pp. 63-74, 1994.

[22] C. Kirchsteiger et al., "Rapid Exploration of Multimedia System-on-chips with Automatically Generated Software Performance Models," ESTImedia, pp. 19-24, 2008.

[23] S. Banerjee, "Macro Pipelining Based Scheduling on High Performance Heterogeneous Multiprocessor Systems." IEEE TSP 43, pp. 1468-1484, 2005.

[24] S. Bakshi, "Hardware/software Partitioning and Pipelining," DAC, pp. 713-716, 1997.

[25] S. Shee et al., "Heterogeneous Multiprocessor Implementations for JPEG: A Case Study," CODES+ISSS, pp. 217-222, 2006.

[26] S. Shee et al., "Design Methodology for Pipelined Heterogeneous Multiprocessor System," DAC, pp. 811-816, 2007.

[27] Ko, D., "The Pipeline Decomposition Tree: An Analysis Tool for Multiprocessor Implementation of Image Processing Applications," CODES+ISSS, pp. 52-57, 2006.

[28] Gordon, M. I., "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs," ASPLOS, pp. 151-162, 2006.

[29] P. Panda et al., "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications." DATE, pp. 7–11, 1997.

[30] M. Verma et al., "Data Partitioning for Maximal Scratchpad Usage," ASP-DAC, pp. 77–83, 2003.

[31] E. Brockmeyer et al., "Layer Assignment Techniques for Low Energy in Multi-Layered Memory Organisations," DATE, pp. 1070-1075, 2003.

[32] M. Kandemir et al., "Dynamic Management of Scratch-pad Memory Space," DAC, pp. 690-695, 2001.

[33] I. Issenin et al., "Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies," DATE, pp. 202-207, 2004.

[34] I. Issenin et al., "Multiprocessor System-on-Chip Data Reuse Analysis for Exploring Customized Memory Hierarchies," DAC, pp. 49-52, 2006.

[35] D. Cho et al., "Compiler Driven Data Layout Optimization for Regular/Irregular Array Access Patterns," ACM LCTES, pp. 41-50, 2008.

[36] R. Szymanek et al., "A Constructive Algorithm for Memory-aware Task Assignment and Scheduling." CODES, pp. 147-152, 2001.

[37] SystemC LRM, May 2005, (ver2.1). http://www.systemc.org

[38] Jouppi et al., "CACTI 5.1," Technical Report, HP Labs, 2008.

[39] SimpleScalar Tool Set, http://www.simplescalar.com/

[40] S. Pasricha, "TLM of SoC with SystemC 2.0," SNUG, 2002.

[41] S. Pasricha et al., "Extending the TLM Approach for Fast Communication Architecture Exploration," DAC, pp. 113-118, 2004.

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on August 08,2010 at 17:55:06 UTC from IEEE Xplore. Restrictions apply.