A Framework for Memory-aware Multimedia Application Mapping on Chip-Multiprocessors

Luis Angel D. Bathen [#1], Nikil D. Dutt [#2], Sudeep Pasricha [*3]

[#] Center for Embedded Computer Systems, University of California, Irvine, CA, USA
{[1] lbathen, [2] dutt}@uci.edu

[*] Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO, USA
[3] sudeep@engr.colostate.edu

Abstract—**The relentless increase in multimedia embedded system application requirements as well as improvements in IC design technology have motivated the deployment of chip multiprocessor (CMP) architectures. Task scheduling and data placement in memory are two of the most important steps in the application customization process as they greatly influence overall power consumption, and performance. Most designers consider task scheduling and data placement to be independent of each other. However, optimal task scheduling does not always produce optimal data placement, and optimal data placement may not necessarily allow for optimal task scheduling. In this paper, we propose a novel framework for simultaneous application mapping and data placement onto CMP architectures, especially for multimedia applications. At the core of our framework is a memory-aware task scheduling algorithm that relies on static analysis and task splitting to reduce off-chip memory transfers. Our experiments on a JPEG2000 case study have shown that we can achieve up to 35% performance improvement and up to 66% power reduction compared to traditional scheduling/data allocation approaches.**

I. Introduction

Power, area, performance, and reliability are among some of the driving factors for embedded systems design. The need for low-power architectures has also increased with the growing demand for battery powered media-rich embedded systems. Along with the demand for low-power embedded systems, scalability limitations in the uniprocessor domain [1], and advances in IC design technology have played a major role in motivating the need for chip-multiprocessors (CMPs). Olukotun et al. [2] showed that CMP platforms perform 50-100% better than superscalar architectures for applications with high levels of parallelism and multiprogramming workloads. 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, 6].

When mapping an application onto an existing CMP platform, task scheduling and data placement are two of the most influential steps as they greatly contribute to overall power consumption, and performance. Current embedded systems have heterogeneous on-chip memory hierarchies composed of 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 [7]. SPMs unlike caches, lack the hardware support to transfer data to/from the different levels of memory, so their management is left to the compilers, which transfer data to/from SPMs by using DMA engines.

Task scheduling has been extensively studied in the field of parallel computing [8]. Most scheduling techniques start with a task graph, and a number of processing cores with large data caches, where the general goal is to minimize the execution time of the task graph. Designers often look at task scheduling and data placement as two separate steps, where they try to optimize each step separately. When moving from the larger parallel computing environment to the CMP domain where resources are not as abundant, most task scheduling techniques fail to explore what data to map into memory, as a given schedule will directly affect what data is brought into memory (SPM). Similarly, the data that is brought into an SPM will affect the CPU's performance, which will in turn affect the current task schedule's execution time. This circular dependency scenario motivates the need to explore different schedules and data placements in order to arrive to the best schedule/data placement solution.

In this paper we propose a framework for memory-aware task mapping on CMPs that tightly couples task scheduling with data mapping onto SPMs. The main contribution of this paper is the development of a framework that will allow designers to explore different memory-aware task schedules, and their data placements, for a given CMP platform, or a series of platforms, where task splitting is performed on each task (when possible), and early task execution is calculated by statically analyzing the task's iteration space, which improves the application's throughput. Our experiments on a JPEG2000 case study show that we can achieve up to 35% performance improvement and up to 66% power reduction over traditional scheduling/data allocation approaches. The rest of the paper is organized as follows. Section II presents an overview of the related work in this area. Section III describes the assumptions and problem formulation. Section IV describes the JPEG2000 case study. Section V discusses our proposed framework. Section VI discusses our experimental results. Finally, Section VII presents the conclusion.

II. Related Work

A. Directed Acyclic Graph (DAG) Task Scheduling

Weighted directed acyclic graph (DAG) task scheduling has been the subject of intensive research work in the field of parallel computing and has been proven to be NP-complete [9]. DAG based scheduling algorithms start with a precedence graph, and attempt to optimally map tasks to a series of processors. Techniques range from very restricted DAG structures [10], unit computational costs [11], and negligible communication costs [12] to algorithms that consider unlimited resources [13]. Most of these algorithms are greedy heuristics that rely on list scheduling, and the tasks' critical path. The highest level first with estimated times (HLFET [14]) algorithm tries to schedule a task to a processor that allows for the earliest start time, where tasks are sorted based on the static bottom-level start time. The insertion scheduling heuristic (ISH [15]) algorithm, much like HLFET, tries to schedule a task as early as possible, but there are instances where processors are idle, so it inserts unscheduled nodes into these time slots. The insertion of tasks into idle slots proves to improve performance significantly [8].

B. *Data Placement/Allocation*

One of the main concerns with SPMs is that the burden of efficiently mapping data is now left to the compiler. Since the 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. [16] profile the application and try 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. [17] look at an application's arrays, and they identify candidate arrays for splitting, if they decide to split an array, they try to find an optimal split point, in order to map the most commonly used areas of the array to SPM, leaving remaining array elements in main memory. Brockmeyer et al. [18], 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 in order to save power. Kandemir et al. [19, 20] 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. Issein et al. [21, 22] 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.

C. *Combined Data Allocation and Task Scheduling*

Although the amount of research in the areas of task scheduling and SPM data allocation is very extensive, there is limited work that tries to address both problems as one. Suhendra et al. [23] 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 have high gains in performance improvement. Szymanek et al. [24] 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. Our approach is different from these techniques in that we perform task partitioning in order to increase the application's throughput, and minimize memory requirements for each subtask. We also introduce the notion of early execution for tasks in order to reduce the end-to-end execution time for the application.

III. Assumptions and Problem Formulation

A. *Architecture*

Figure 1 shows our architectural model, which is similar to the one used in [23]. 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, where 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.

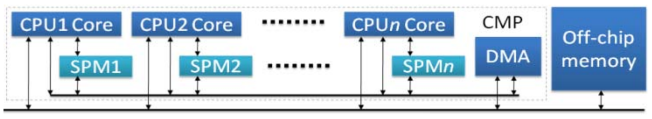

Fig. 1. CMP Architecture

B. *Assumptions*

Our approach assumes that the designer has already partitioned the application into a series of tasks (nodes) and their dependences (edges). We also assume that designers are aware of the critical computational kernels among each task, and they can extract and abstract the information as a series of well defined loop nests, which contain array accesses with affine loop expressions. Issenin et al. [25] proposed a method to automatically generate affine functions for loop nests from original C programs. Our framework also requires designers to estimate the initial execution time for each task through profiling for the exploration step.

C. *Problem Formulation*

Existing scheduling and data placement techniques are often considered two separate steps in the design process. It has been shown that by combining these two problems into one can drastically improve performance [23, 24]. However, these techniques rely solely on profiling information to determine what data to place in memory. Static analysis techniques such as the ones presented in [19-22] show that by statically analyzing the application they can discover new opportunities for improvement, be it loop transformations to improve locality, or data reuse analysis. The main goal of our framework is to enable the exploration of different memory-aware task schedules and their data placements for a given application to minimize off-chip data accesses, in order to reduce power dissipation and improve application performance. Our framework relies on application profiling in order to determine initial task execution times, as well as static analysis to identify split points for tasks, reuse and data dependencies across tasks, and data placement information.

IV. Case Study: JPEG2000

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), where each filter performs a different function on the data. Most data reuse techniques focus on inter-task reuse, whereas we focus on intra-task reuse. The assumption is that each task operates over the same data (intra-task data reuse), performing different functions. Due to the amount of data that multimedia applications must process, it is common practice to partition the data and process it independently. The idea of intra-task data reuse can be observed by looking at the data flow in the JPEG2000 [3] encoder from Figure 2.

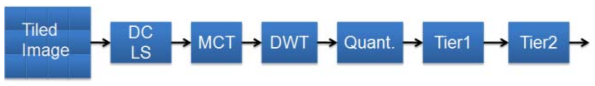

Fig. 2. JPEG2000 Encoder Block Diagram

The image goes through a preprocessing phase that may include tiling (splitting the image into smaller images called tiles, which are processed independently), formatting the pixel value ranges, and multi-component transformation. Next, each component is processed by the DWT filter, which decomposes the image into multiple subbands at different resolution levels, where each subband (HH, HL, and LH) represent a downsampled residual representation of the image, and the LL subband represents a 2:1 subsampled 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 (optional). The array now contains quantized values which are then passed to the Tier1, which performs bit plane coding and arithmetic coding before generating the final encoded bit stream.

V. Proposed Framework

A. Overview

Our framework consists of three main components, the first being our simulated annealing (SA) [27] exploration engine, which allows us to explore different schedules, data placements for a given platform or a series of platforms, as well as to how many tasks to attempt to execute in parallel as this might impact the amount of thrashing/idle cycles processors will have. The second component is our memory-aware scheduler which tries to find task splitting opportunities by examining the tasks' loop nest, and schedules tasks with the goal of minimizing DMA transfers and improving the application's throughput. The last component is the modeling and evaluation of the final *N* schedules and data placements.

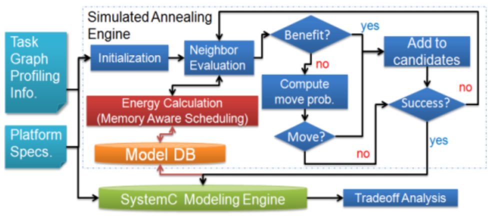

Fig. 3. Proposed Framework

Figure 3 shows the block diagram of our framework. Our framework takes in two inputs from designers. The first is the task information, which consists of the initial task graph and the critical computation kernels for each task. The second input is the platform being targeted and the number of candidate configurations to be examined during the tradeoff analysis stage. Designers have the option of allowing the exploration engine to explore different platforms consisting of varying CPU/SPM size configurations, by providing upper and lower bounds for both entries. The first step in the exploration process is the initialization of the SA algorithm, which consists of the platform model generation, temperature setup, initial state (where each state consists of a task graph, and a platform configuration), and the state's initial energy. The state's energy is obtained by evaluating the performance (execution cycles) of the memory-aware schedule and its corresponding data placement. The next step is the selection of a neighbor state, which is evaluated and compared with the current state. If there is substantial benefit, we add it to the possible candidates. If there is no benefit, we compute the move acceptance probability, and decide whether or not to move to the state. This keeps us from falling into the local optima trap. After every move to a new state, we update the number of successful swaps. Our framework continues to evaluate different states until the limit of successful swaps is reached. The SystemC [26] [29,30] modeling engine allows us to model the best *N* candidates passed down from the SA engine and evaluate them in terms of end-to-end execution cycles, power consumption, CPU execution/stall cycles, etc. This final step will help designers choose which among the different configurations serves their needs best.

B. Task Splitting

In this work, we will consider a straight forward type of task splitting which will rely on splitting the address space dedicated to each task into independent continuous address regions so that each task can be executed independently. This technique also allows us to take full advantage of DMA bursts as short bursts and resets may lead to performance degradation. The splitting of tasks will be determined by the amount of available SPM space the designer wishes to allocate to our technique. Since we are looking at tasks from a coarser level of granularity, it is possible to combine our approach with finer level compiler optimization techniques. As part of our future work, we plan on extending our framework to consider loop transformations and inter-task data reuse analysis [21, 22] as part of the exploration process, as we are aware that different loop transformations will affect our data placements/schedules.

Our approach is split into three main parts, the generation of the hierarchical set tree, the actual splitting of tasks, and finally, the generation of *early execution edges*. We introduce the notion of a hierarchical set tree, which is built bottom-up from the inner-most loop to the outer-most loop in the nest. Each loop level represents a set S consisting of the union of the sets in its lower loop level. Set *S* at loop level *i* denoted by S_i is a set of the clustered array accesses at loop level *j*, (S_j) this can be represented by:

$$S_i = \bigcup_{l}^{u} S_j, \forall i < j, i, j \in N$$

Where *u* is the loop's upper bound at loop level *i*, and *l* is the loop's lower bound at loop level *i*. Note that *u* and *l* can also represent a range in the iterator space which is a subset of the range represented by the loops' upper and lower bounds. The loops' levels are determined top-down, meaning that the outer-most loop will have level *0*, and the inner-most loop level *n-1*, where *n* is the number of loop nests. Figure 4 shows a loop nest consisting of four levels. Each level consists of a set of sets of array accesses, which are represented by the set's *start* and *end* points. Level 3 can be viewed as a set of linear array accesses (1x32 elements), which can be viewed as a leaf node in the tree. Level 2 consists of a set of level 3 accesses (a 32x32 block), where each child node consists of 1x32 elements. Level 1 consists of a set of level 2 accesses (a 4x32x32 block), where each child consists of 32x32 blocks. Finally, at the root of the tree, we have Level 0, which contains 4 children nodes, each consisting of 4x32x32 blocks.

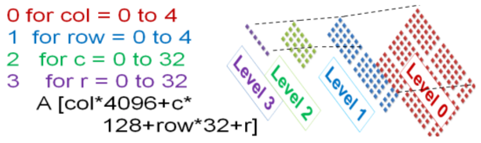
```
0 for col = 0 to 4
1  for row = 0 to 4
2   for c = 0 to 32
3    for r = 0 to 32
      A [col*4096+c*
              128+row*32+r]
```

Fig. 4. Hierarchical set tree for each loop level

Once we have generated the hierarchical set tree we can start the task splitting procedure. We split the sets into a list of independent sets, where any two data sets within a loop level are independent when their intersection is *null*. If the intersection doesn't return a *null* set, then we create a new set *x*, where $x \in S_i \cup S_j$. We remove both S_i, and S_j and insert set *x* into the list of sets for further merging. At the end of this process, we will have a list of independent sets. The next step is to find a set of maximum sets (S_{imax}), where each maximum set (S_{max}) is list of independent sets that satisfy the following condition:

$$S_{imax} = \bigcup_{i}^{n} S_i, s.t. \sum_{i}^{n} size(S_i) \leq SPM, 0 \leq i \leq n$$

Once we have generated S_{max} we proceed to split the loop into independent loops that operate over independent data sets. For every set S_{imax} in S_{max} we clone loop *L,* and create a new loop node (*L_clone_i*), where this new loop's iterator (*iterator_copy_i*) will iterate over a subset of loop *L's* iteration space determined by the *start* and *end* points for the current maximum set S_{imax}. We also update the clone's hierarchical tree of sets, which will point to the root node (root set) in S_{imax}. Consider the loop nest from Figure 4, where we are walking an array of 128x128 elements, and processing blocks of 32x32 pixels, assuming that we have 8KB of SPM space available. Our technique will allow us to split the outermost loop into two independent loop nests, with copy iterators *col_copy_1* and *col_copy_2* going from *0* to*1* and *2* to *3* respectively, where each cloned loop nest will walk over half of the iteration space.

We introduce a second type of dependence edges into the task graph called *early execution edges*, which allow our scheduling heuristic to determine whether or not it is possible to start a task before its dependence finishes its entire execution. We obtain *early execution edges* by calculating the live-range for a set of array accesses, so we can identify at which iteration the given set is no longer referenced. When we split a task, each of its subtasks is assigned a branch of the hierarchical set tree from its original task, so that each subtask can be executed independently. For each task T_j in T_i's dependence list, we take T_i's set of accesses S_i and search for an overlapping set in the current dependence task's set of accesses S_j. If we find an intersection, we compute the iteration in T_j when this overlap occurred, there could potentially be more than one occurrence, so we keep track only of the last occurrence. The goal is to identify the last iteration in task T_j when the overlap occurred. Once we find this iteration, we create an *early execution edge* from task T_j to T_i containing the triplet <*loop level, iterator, iteration*>.

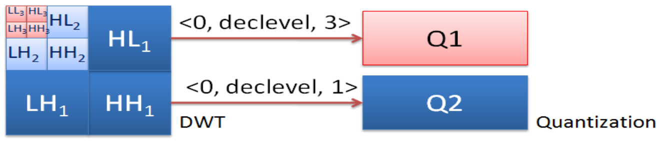

Fig. 5. Task splitting for DWT and Quantization

Consider JPEG2000, where the quantization task depends on the DWT coefficients from the DWT task. The main DWT computational kernel consists of a triple nested loop, where the two inner loops perform the 2D-wavelet transform on each row/column, and the outermost loop determines the level of decomposition. Since quantization operates over each DWT subband independently, we can split the quantization task into two independent tasks, Q1 and Q2 as shown in Figure 5, where Q2 operates over LH_1 and HH_1, and Q1 operates over the remaining decomposition levels. By splitting the tasks we can speed the encoding process by starting quantization task Q2 as soon DWT produces the subbands for the first decomposition level. To represent this possibility during our scheduling step, we introduce two *early execution edges* between the DWT task and both tasks, Q1 and Q2. The *early execution edge* between DWT and Q2 tells the scheduler that Q2 can start execution when DWT's iterator *declevel*, declared in loop level 0, is equal to 1.

C. Memory-aware Task Scheduling

Our heuristic follows the basic idea of the insertion list scheduling as discussed in [15] which tries to find idle slots in the current schedule and populate them. We start with a list of tasks to schedule. Figure 6 shows a high level overview of our heuristic. First, all tasks are sorted in descending order based on their execution time. Tasks with lower execution time are more flexible when it comes to their scheduling as their impact on the critical path is not as great as tasks with long execution times. We then proceed to build our schedule one task at a time. We get the next available CPU *p* and task *t* from the ordered task list and check to see if all of its dependencies are done executing. If *t's* dependencies have not completed their execution, we check to see if there are *early execution edges* for *t*. For every *early execution edge* going to *t*, we uses the edge's information to determine if it is possible to execute the task *t* 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 *t*. Before we map task *t*, we look at the data currently placed in *p's* SPM, and search for a task *alt* which depends on this data. If we find such task, we map it to *p*, otherwise, we map task *t* to *p* at the cost of a DMA transfer. This helps us keep the number of DMA transfer to a minimum.

```
// memoryAwareTaskScheduling(T=G(V,E))
sort all tasks in descending order (based on their execution time)
while there are unscheduled tasks in T do
    for each unscheduled task t in T do
        cpu = next available cpu
        if(allDepsFinished(t) or
            canStartEarlyExecutionNow(t)) then
            task alt = getAltTaskToMap(cpu, tasks)
            if(alt!=null) then
                map(alt,cpu)
                remove alt from T
            else
                create dma xfer for task t
                currTime = currTime + getCommDMACost(t)
                map(t,cpu)
                remove t from T
```

Fig. 6. Memory-aware task scheduling algorithm

VI. Experimental Results

A. Experimental Setup

Our experimental setup consists of a base case where designers start with a task graph, and try to schedule it using the insertion scheduling heuristic [15]. The most commonly used data is then fetched and stored in local SPM, in the case of JPEG2000, the data in the LL subband is given priority as it is the most reused set within the DWT kernel. The base case approach executes *N* task sets (tiles) in parallel, so at any given moment there are a total of *N* tiles being processed by the *N*-CPU CMP. Performance is measured in terms of end-to-end execution cycles it takes to process an image on our bus level cycle accurate SystemC models. Since we are trying to minimize off-chip and DMA memory transfers, our main concern is dynamic power. We obtained power estimates by using the CACTI v5.2 toolset [28].

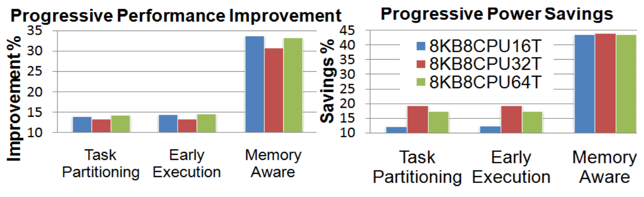

Fig. 7. Progressive performance improvement

B. Results

We compare our memory aware scheduling in a progressive approach with three other cases. The first one is the base case, the second approach incorporates our task partitioning technique (TP), and the third approach incorporates our task partitioning and *early execution edges* techniques (EE). Figure 7 shows performance improvement for the two progressive approaches (TP and EE) as well as our technique which is a combination of TP, EE and our memory-aware scheme compared to the base case with an 8KB/8CPUs configuration, and 16, 32, and 64 task sets being scheduled at the same time. The positive effects of our approach include higher throughput due to increased task level parallelism and load balancing as more CPUs are busy, as well as being able to fit more task specific data on local SPMs. On the other hand, increasing the number of tasks may lead to an increased amount of stall cycles due to increased number of dependencies, main memory contention, and thrashing, as every task wants to bring its own data to local SPM. Since our algorithm tries to bring tasks to where data is, rather than data to where tasks are, we minimize thrashing.

C. Impact of increasing simultaneously scheduled tasks

There is a need to explore how many task sets we wish to schedule at a time. As our insertion based scheduler tries to schedule tasks on idle CPUs, how many tasks to look ahead (how many tiles to fetch and schedule) has an impact on the schedule's performance. Ideally, scheduling all task sets (tiles) at the same time would minimize idle times, however, this might not always be the case since aggressively inserting to maximize CPU utilization might increase the thrashing effect as well as the stalls due to dependencies.

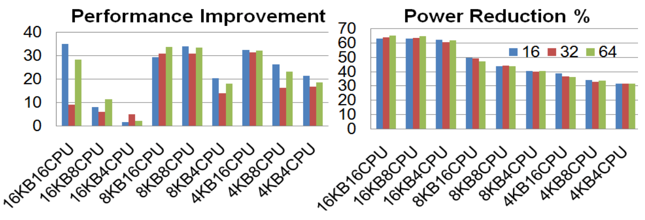

Fig. 8. Scheduling multiple tiles simultaneously

Figure 8 shows that there are cases when scheduling more task sets at the same time is not as beneficial, given the platform being targeted. For instance, for a CMP with 16KB/16CPUs it would be better to schedule 64 task sets, meaning, process 64 tiles at a time, than schedule 16, or 32 task sets. On the other hand, if a platform had less CPUs and SPM space (4KB, 4CPUs), scheduling 16 task sets at a time would be more beneficial. One could guess, for smaller platforms, schedule less tasks, and for bigger platforms, schedule more, however, this is not always the best approach as we can see in the 8KB/16CPUs and 16KB/4CPUs configurations, hence, the need for exploration.

D. Simulated Annealing Performance

One of the main questions when it comes to exploration heuristics such as simulated annealing is how good with respect to the optimal case are they. In these set of experiments we measured the performance of our simulated annealing technique for multiple successful swaps. We compared our results with the optimal solution obtained by exploring the entire search space over a period of nine hours. The search space included a total of 131072 (1-32CPUs, 1-32KB SPM size, 1-128 task sets, each task set consists of 3 tasks) states.

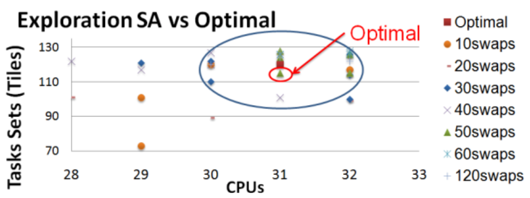

Fig. 9. Simulated Annealing vs. Exhaustive Optimal

Figure 9 shows that as we increase the number of successful swaps, our chances of arriving to the optimal solution increase. If we are to expand the search space to say 1-128CPUs, it would quadruple our search space, resulting in days of simulations, where as if we were to use our simulated annealing approach, if we were to choose a standard deviation between 2 and 5%, we would have results within two hours.

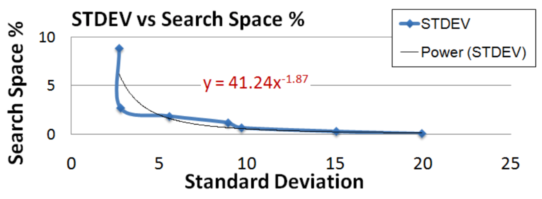

Fig. 10. Standard Deviation vs. Search Space

Figure 10 shows the percentage of the search explored as a function of the standard deviation. A standard deviation between 2 and 5% would require our algorithm to explore a total of 2% of the search space, we could further relax our requirements, to allow anywhere between 1-10% standard deviation from the optimal solution, which would explore anywhere between 0.05 to 2% of the search space.

VII. Conclusion

In this paper we introduced a framework that enables memory-aware task scheduling for CMP platforms. Our framework allows designers to explore and compare different task schedules, SPM data placements, and platform configurations (SPM sizes and number of cores) for a given application. We also introduce a task splitting mechanism that improves the application's throughput by increasing the task level of parallelism. We augment an existing scheduling heuristics to include the concept of early execution time by introducing *early execution edges* into the task graph. Our experiments on a JPEG2000 case study have shown that we can achieve up to 35% performance improvement and up to 66% power reduction compared to standard scheduling/data allocation approaches.

Acknowledgements

This work is supported in part by the Eugene Cota-Robles Fellowship from the University of California, Irvine.

References

[1] M. S. Hrishikesh et al. "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," *ISCA* , p. 248, 2000
[2] K. Olukotun et al. "The case for a single-chip multiprocessor." *SIGPLAN*, pp. 2-11. Sep. 1996.
[3] *JPEG2000*, ISO standard, ISO/IEC 15444-1, 2000.
[4] *ITU-T Rec. H.264*, ISO/IEC 14496-10, 2003.
[5] S. Sun et al. "A Highly Efficient Parallel Algorithm for H.264 Encoder Based on Macro-Block Region Partition." *HPCC 2007*, pp. 577-585
[6] J. Chong et al. "Efficient Parallelization of H.264 Decoding with Macro Block Level Scheduling." *ICME*, pp. 1874-1877, July 2007.
[7] R. Banakar et al. "Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems," *CODES*, 2002.
[8] Y. Kwok , I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms", *JPDC*, v.59 n.3, p.381-422, Dec. 1999.
[9] M. Garey et al. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman and Co., 1979
[10] C. Papadimitriou and M. Yannakakis, "Scheduling Interval-Ordered Tasks," *SIAM J. Computing*, 8, 1979, pp. 405-409.
[11] T. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 19, no. 6, pp. 841-848, Nov. 1961.
[12] C.V. Ramamoorthy et al. "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Trans. Comp.* C-21, pp. 137-146, 1972.
[13] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *ICPP*, pp. 1-8, 1988
[14] T. Adam et al. "A Comparison of List Scheduling for Parallel Processing Systems," *Comm. ACM*, v17, n12, pp. 685-690, Dec. 1974.
[15] B. Kruatrachue et al. "Duplication Scheduling Heuristics (DSH): A New Precedence Task Scheduler for Parallel Processor Systems," *TR*, Oregon State University, Corvallis, OR 97331, 1987.
[16] P. Panda et al. "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications." *DATE,* pp. 7–11, 1997.
[17] M. Verma et al. "Data Partitioning for Maximal Scratchpad Usage", *ASP-DAC*, pp. 77–83, 2003.
[18] E. Brockmeyer et al. "Layer Assignment techniques for Low Energy in Multi-Layered Memory Organisations," *DATE*, pp. 1070-1075, 2003.
[19] M. Kandemir et al. "Dynamic management of scratch-pad memory space," *DAC*, 2001
[20] M. Kandemir et al. "Compiler-Directed scratch pad memory hierarchy design and management," *DAC*, 2002
[21] I. Issenin et al. "Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies," *In Proceedings of DATE*, 2004.
[22] I. Issenin et al. "Multiprocessor System-on-Chip Data Reuse Analysis for Exploring Customized Memory Hierarchies," *DAC*, 2006.
[23] V. Suhendra et al. "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures", *CASES*, 2006.
[24] R. Szymanek et al. "A constructive algorithm for memory-aware task assignment and scheduling." *CODES*, pp. 147-152, 2001
[25] I. Issenin, Nikil Dutt. "FORAY-GEN: Automatic Generation of Affine Functions for Memory Optimizations," *DATE*, pp. 808-813, 2005
[26] SystemC LRM, May 2005, (ver2.1). http://www.systemc.org
[27] S. Kirkpatrick et al. "Optimization by Simulated Annealing," *Science*, Vol 220, Number 4598, pages 671-680, 1983.
[28] Jouppi et al. "CACTI 5.1", *Technical Report*, HP Labs, 2008
[29] S. Pasricha, "TLM of SoC with SystemC 2.0" SNUG 2002
[30] S. Pasricha et al., "Extending the TLM Approach for Fast Communication Architecture Exploration", DAC 2004