A Methodology for Power-aware Pipelining via High-Level Performance Model Evaluations

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— **Power is one of the major constraints considered during the design of embedded software. In order to reduce power consumption without sacrificing performance, software needs to be optimized in order to run as efficiently as possible on a given platform. When attempting to optimize the mapping of a piece of software on a multiprocessor system, designers often face the chicken-and-egg problem of whether to schedule tasks first, or do memory allocation first, as either step will affect the different optimization opportunities the other may provide. Because each optimization will affect the system's power consumption, it is critically important to be able to monitor the effects these transformations have. In this paper we present a methodology that allows designers to quickly evaluate the impact each code optimization will have in the system's power. Our exploration engine relies on SystemC-based power/performance models to quickly and accurately evaluate the dynamic power due to memory accesses as well as the expected CPU power consumption.**

I. Introduction

With the ever growing demand for media-rich data intensive applications, the need to push for high performance, yet low power solutions is ever more present. Recent studies have shown previously, chip-multiprocessors (CMPs) are ideal platforms for applications with high levels of parallelism [1]. Examples of such applications are multimedia applications such as JPEG2000 [2] and H.264 [3]. As application designers move towards chip-multiprocessor systems leaving the old high performance uniprocessor domain, new challenges arise. Among these challenges, one of the most difficult tasks is how to map an application onto a multicore system. During the application mapping process, task scheduling, and data allocation are two of the most critical steps. Tightly coupling these two steps is critical as each schedule will give us different data allocation opportunities, similarly, each data allocation will affect the schedule's throughput. Because the exploration space is quite large as a small change in a schedule will impact what data needs to be mapped to which memory location, there is a need for fast yet accurate exploration engines.

The most important part of an exploration engine is its cost function, be it performance, or power, whatever it is what designers are trying to optimize for, the cost function will determine whether the exploration will yield good or bad results. Recent work [4] looked at enabling exploration of the search space by statically evaluating the cost function given by the number of on-chip/off-chip accesses, which was used to derive both throughput as well as dynamic power estimates.

In this paper we present an exploration engine that relies on SystemC-based power/performance simulation models to quickly and accurately evaluate the dynamic power due to memory accesses as well as the expected CPU power consumption. Our methodology attempts to optimize the application in order to improve its performance and reduce its power consumption by applying loop-level transformations, task-level pipelining, as well as data reuse analysis, generating their respective performance and power models. The main contribution of this work is the underlying framework that allows us to rapidly generate and explore different memory-aware pipelined schedules through system-level power estimation with different levels of granularity.

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 covers related work. Section VII presents experimental results. Finally, section VIII presents concluding remarks and future work.

II. Motivations and Contributions

A. Multimedia Applications: JPEG2000 Case Study

Our methodology targets media-rich data intensive applications, with both data level as well as task level parallelism. Due to the amount of data that these type applications process, it is very common to partition the data and process it independently.

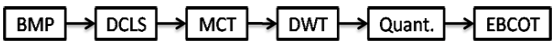

Figure 1. JPEG2000 Block Diagram.

Figure 1 shows the data flow in the JPEG2000 encoder, where the image goes through a preprocessing phase that formats the pixel value ranges (DCLS), decomposes color domains (MCT), and splits images into smaller blocks (tiles). Each tile component is processed by the discrete wavelet transform (DWT), which decomposes the image into multiple subbands at different resolution levels, where each subband (HH, HL, and LH) represents a

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

1550-4093/09 $26.00 © 2009 IEEE
DOI 10.1109/MTV.2009.19

down-sampled residual representation of the image, and the LL subband represents a 2:1 sub-sampled version of the original image. DWT coefficients go through the scalar quantization step improving the compression rate at the loss of quality. Finally, quantized values are entropy coded (EBCOT) before generating the final bitstream.

B. Inter-kernel Data Reuse

Most data reuse techniques [5-7] that focus on intra-kernel reuse look at a single kernel, and try to exploit its reuse without considering the impact this will have on the other kernels. Similarly, classical task scheduling approaches focus on mapping tasks without considering the notion of data reuse. Each task might contain a series of computational kernels, where data for each task is mapped independently of whether the current task re-uses data from the previous task or not.

```
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] = G[i][j]  - p(2, s-1)
    R[i][j] = R[i][j]  - p(2, s-1)
  }
```
(a)

```
void mct
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.

Figure 2 shows two tasks, DCLS (a), and the multi-component transform MCT (b). As we can see, MCT depends on the data [R, G, B arrays] produced by DCLS. Standard scheduling would schedule the execution of DCLS, followed by MCT, where 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 with little benefit to be gained from data-reuse techniques such as [6, 7]. However, the amount of inter-kernel reuse is significant (italicized/bold arrays). Realizing that each of MCT's components will utilize at least two data streams produced by DCLS can reduce the number of unnecessary data transfers.

C. Need for High-Level Estimation

Power consumption is one of the most critical requirements in embedded systems today. Because of its importance, it is extremely necessary to estimate power consumption in the early stages of a system design. As we move towards the multiprocessor domain, mapping an application is challenging because each design/mapping decision can have an impact on how much power the application will consume on the given platform.

D. Contributions

In this paper we propose a methodology for power and performance model generation, and their use for evaluating code optimizations. Our main goal is to generate low power schedules and data allocation schemes without sacrificing performance. We are able to generate performance models from each of our pipelined schedules, and evaluate their performance at different levels of accuracy, depending on how much detail designers want. Designers might choose to look at the impact our kernel level transformations and reuse analysis has on the data cache, scratchpad memory (SPM), or how the partitioning of kernels affects the instruction cache.

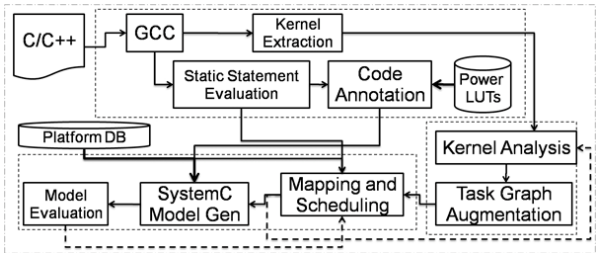

Figure 3. Methodology Overview.

III. Proposed Methodology

A. Overview

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 can be composed of any number of computational kernels, which might be further decomposed into a series of individual pipelined computational kernels by our methodology.

Our methodology is divided into three main steps: *(i)* a front end, which is responsible for generating input performance 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. These steps are described in the following subsections. The dashed edges represent feedback edges which help our methodology refine scheduling options as well as transformation types.

B. Performance Model Generation

The first step in our approach is to generate the necessary input performance 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. We use the Simplescalar tool set [8] to generate our target assembly as well as our initial profiling information which includes execution time for each task and initial task dependencies. This information is then used to annotate back the functional model with timing delays as well as power consumption counters for the simulation model generation. Each pipelined schedule instance will have a different data and task mapping, so it is extremely important to be able to see the impact on system power each of our optimizations has. Our framework allows us to explore a large search space by evaluating each of our schedule instances through system-level performance models built on top of SystemC 2.0 [9]. Figure 4 shows a high level overview of the performance model generation process. The models start as functional models which are then subjected through a series of steps that include kernel

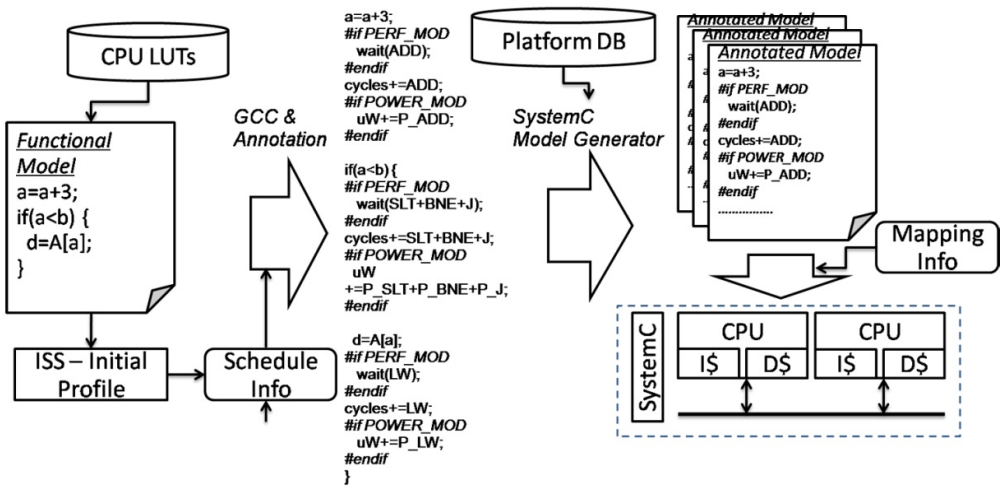

Figure 4. Performance Model Generation Process.

extraction, and source level optimizations such as loop tiling, fission, etc. Once kernels have been optimized, we compile them and insert delay information. We also insert instruction counters as well as power counters obtained by means of a processor look-up-table (LUT) to estimate both power and execution time. Once each kernel has been annotated, it goes through the SystemC model generation which generates the CMPs according to the designer's specifications. Similarly, our memory-aware pipelining heuristic generates a schedule and data mapping, which can be evaluated at different levels of accuracy.

C. Code Transformations and Task Graph Generation

void dcls_tiled	***void mct_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+=tw) for(i=ii; i<min(m, ii+tw); i++) for(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(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+=tw) for(i=ii; i<min(m, ii+tw); i++) for(j=jj+i; j<min(n+i, jj+tw+i); j++) Ur[i][j-i] = *B[i][j-i]* - **G[i][j-i];**
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)***	for(ii=0; ii<m; ii+=tw) for(jj=0; jj<n; jj+=tw) for(i=ii; i<min(m, ii+tw); i++) for(j=jj+i; j<min(n+i, jj+tw+i); j++) Vr[i][j-i] = ***R[i][j-i]*** - **G[i][j-i];**
(a)	(b)

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

Designers subject the application to a series of loop transformations such as loop fission to decompose large kernels o smaller and possibly independent kernels, where each kernel operates over a different data set. Loop tiling is utilized to break down the computation of large data sets o smaller units of computation, and loop unrolling is utilized to further decompose the execution of kernels. By gradually applying loop transformations to the code, we can partition the execution of each kernel o smaller computational kernels, and pipeline them on a multiprocessor system fully utilizing the available resources. For a detailed description refer to [4]. Each kernel produced can be represented as a node in our augmented task graph, where its dependence edges are determined by the order of execution as well as its data dependencies. We also exploit the idea of early execution edges [4] which allows us to find earlier execution of tasks thereby improving performance. Figure 5 shows the tiled versions of the two tasks in Figure 2 which shows how we have gone from two kernels to six kernels, each with a series of inter-kernel data dependencies.

IV. Target Platforms and Assumptions

A. Target Platforms

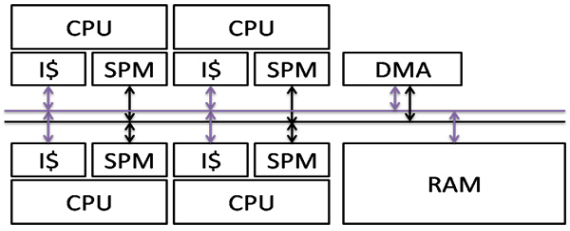

Figure 6. Homogeneous CMP Platform

We consider simple homogeneous CMP platforms, consisting of multiple processing cores, each with its own SPM/data cache, instruction cache, and a DMA engine to facilitate the data transfers among the various SPMs. Our CMPs make use of the shared bus communication infrastructure since they are still the most dominant types of communication fabrics used in embedded systems. In the case of the SPM based CMP, each processing core 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. Similarly, the cache based CMP has a similar structure as the SPM based CMP, with the distinction that the cache based CMP replaces a cache with a SPM of equal size. Figure 6 shows a 4 Core homogeneous CMP with SPMs replacing the data

cache. These types of CMPs are particularly useful for applications with large levels of parallelism. In particular, if access patterns are regular enough, the SPM based system would be preferred as the application's access patterns can be analyzed, thereby, allowing designers to efficiently manage the platform's memory.

B. Assumptions

Our methodology assumes that the designer has already partitioned the application into a series of tasks (nodes) and their initial dependences (edges) have been resolved. 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. Tasks with high levels or irregularity are still part of our task graphs, but are omitted from the code transformation path. We also assume that designers have been able to generate power LUTs for the given processing core they wish to use for the performance models.

V. Mapping and Scheduling

In this paper, we use the quantum-inspired evolutionary algorithm (QEA) to map tasks onto multiprocessors [11]. QEA has the advantage of obtaining a set of best alternative solutions at the same time under multiple criteria.

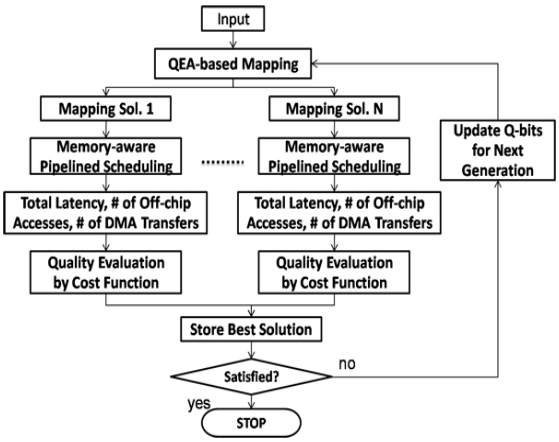

Figure 7. Overall Mapping and Scheduling Flow.

Figure 7 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 instead of binary, numeric, or symbolic representation.

The generated mapping solutions are evaluated by our memory aware pipelined scheduler. Our memory-aware pipelined scheduling heuristic is based on the idea of retiming [12]. 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) [13]. 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) o 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. 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.

Each possible task and data mapping is evaluated by our performance model estimation. Our exploration engine can calculate the cost function at different levels of detail.

VI. Related Work

A. Memory Aware Heuristics

Literature in task scheduling as well as data placement is very mature [14, 5-7]. However, heuristics that attempt to couple both problems as one is limited. Suhendra et al. [15] propose an ILP formulation that combines task scheduling, SPM partitioning, and data allocation. They show that by combining task scheduling and data placement o a single problem, they can achieve high gains in performance. Szymanek et al. [16] 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 o 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. We do not rely entirely on initial profiling information as each schedule and data mapping will give different memory utilizations as well as throughput. We employ our high-level power and performance models to help us drive our exploration engine.

B. Performance Model Generation

Application designers must have a means to rapidly prototype and evaluate the performance of their applications on a target platform. Tools like Simplescalar allow for such prototyping framework, which include a cross-compiler, a cache simulator, as well as a full instruction set simulator (ISS) among other tools. However, as we move towards multicore systems, evaluation of software on a series of ISSs might be an overkill, especially if designers wish to do a quick estimation of power or performance. Moreover, in order to evaluate system-wide performance, and power, designers would have to write wrappers for the ISSs to talk to each other over a communication fabric. In some cases, the RTL/ISS for the target platform might not exist yet, affecting software development. The notion of performance models have been presented before [17]. Performance models allows software designers to quickly evaluate the performance of their software by simply looking at the data sheet of the target platform, writing a LUT, and letting a synthesis tool annotate their software for evaluation. The main difference between our work and [17] is that we are not only interested in looking at performance, but we are also looking to estimate power. We enhance the work presented in [17] by adding power estimation at different levels of the system, from the cores by using a similar LUT based approach, as well as power estimation for the different memory types and buses.

C. Power Estimation

Tiwari et al. [18] were among the first to look at early power estimation of software. They looked at a processor's data sheet, and derived LUTs, and estimating the power at the instruction level. A more recent approach [19] derived 3D LUTs from a processor data sheet, and used them to estimate power consumption at different levels of granularity. From very low level, instruction and functional block power consumption to very coarse grain, processor idle/active states. Our power estimation model relies on two main pieces of work, the work presented by [19] for processor based power estimation and [20] for the communication infrastructure. To explore the impact our methodology has on the memory subsystem we use CACTI [10]. Combining these three pieces of work and incorporating them o our performance model generation allows us to quickly estimate system-wide as well as component-based power estimation. Bansal et al., [21] proposed a framework that incorporated heterogeneous power models for system-level power estimation. The main difference in our work from theirs is that we automatically generate the power models from transformed functional models, which allows us to rapidly explore the design search space. Our main goal is to use our system-level power estimators to drive our scheduling and data allocation decisions while providing the designers with different levels of details (and speed) for the cost functions.

VII. Experimental Results

A. Experimental Setup

Our performance models are built on top of the SystemC 2.0 [9] library. To evaluate the dynamic power due to memory accesses we used the CACTI [10] tool set assuming 65nm technology. We used the Simplescalar tool set [8] to generate assembly code for each task as well as to calculate execution time for each task and data/instruction cache statistics. Our bus models are built on top of our TLM framework [20, 22] and our CPU power models are based on [19]. For our studies we use the JPEG2000 encoder as the case study. To be able to see a difference in the power consumption in the different platform configurations we normalize the average power consumption of each of our examples. Each configuration data point represents an SPM size and the size of the CMP, for example, 16x4 represents a 4 CPU CMP with 16KB SPM size.

B. Choosing the Right Cost Function

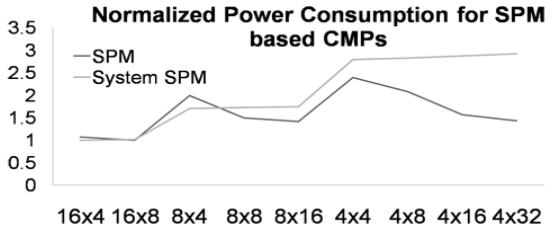

Figure 8. Normalized Average Power Vs. CMP (SPM Size in KB x # CPUs).

Figure 8 shows the normalized average power for a series of platform configurations with respect to a 16KB 8CPU CMP. Choosing the right level of detail for a cost function is critical as the exploration engine's decisions depend on

how good the cost function is. As seen in Figure 8, despite the fact that the SPM power estimates allowed us to differentiate between platform configurations, their system wide power estimates are not consistent as there are other factors that affect overall system-level power, such as the instruction cache, the processors, and the communication infrastructure. In some cases however, if the designer knows that the application is a data intensive application and the stress will be placed on the memories then he/she may choose to take a look at power consumption at the memories as the cost function for exploration. Similarly, if the designer knows that the application being mapped is a control intensive application, he/she may choose to look at CPU wise power consumption.

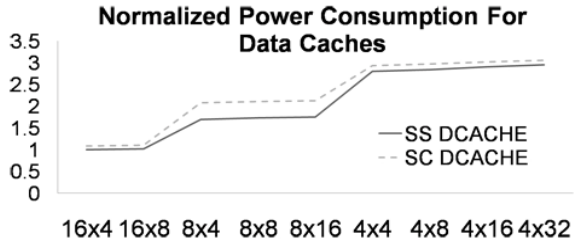

Figure 9. Normalized System Power Vs. CMP (SPM Size in KB x # CPUs).

Figure 9, the normalized power profile for a series of CMPs consisting of hybrid Simplescalar/SystemC performance models (SS DCACHE) and a SystemC only performance models (SC DCACHE). As we can see, both models yield similar power profiles, which show the flexibility of our methodology that allows us to decide which combination of performance models to use to capture system wide power.

C. *Algorithm Runtime*

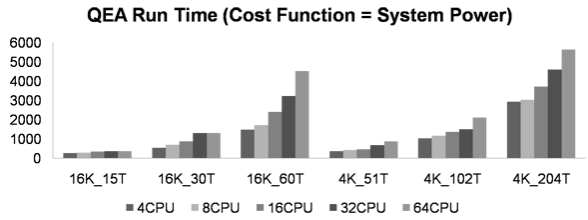

Figure 10. QEA Runtime (seconds) Vs. # Tasks Pipelined.

Figure 10 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 253 seconds for the 16KB 4CPU CMP and unrolling degree of 1 to about 1.5 hours for the 4KB 64CPU CMP and unrolling degree of 4. This shows just a 22x 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.

VIII. Conclusion And Future Work

In this paper we introduced a methodology for power-aware pipelining on chip-multiprocessors. Our exploration engine allows designers to choose the level of detail in their cost functions, be it performance or power, or both. Our methodology relies on source level optimizations to exploit the application's data reuse as well as parallelization opportunities. Because each change in the application may affect the power profile and throughput of the given platform, our methodology helps designers identify which optimizations yield best results. Designers can also walk the search space in as little as 8 seconds, assuming a simple cost function of off-chip memory accesses and throughput, to up to 1.5 hours for a full system-wide power profile as the cost function.

References

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

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

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

[4] L. Bathen et al., "Inter-kernel Data Reuse and Pipelining on Chip-Multiprocessors for Multimedia Applications," ESTImedia, pp. 45-54, 2009.

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

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

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

[8] Simplescalar Tool Set, http://www.simplescalar.com

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

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

[11] Y. Ahn et al., "SoCDAL: System-on-chip Design Accelerator," ACM TODAES, v. 13, pp. 1-38, 2008.

[12] K. Chatha et al., "RECOD: A Retiming Heuristic to Optimize Resource and Memory Utilization in HW/SW Codesign," CODES, pp. 139-144, 1998.

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

[14] Y. Kwok ea al., "Benchmarking and Comparison of the Task Graph Scheduling Algorithms", JPDC, v.59, pp. 381-422, 1999.

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

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

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

[18] V. Tiwari et al., "Power Analysis of Embedded Software: a First Step towards Software Power Minimization," ICCAD, pp. 384-390, 1994.

[19] Y. Park et al., "Methodology for Multi-Granularity Embedded Processor Power Model Generation for an ESL Design Flow," CODES+ISSS, pp. 255-260, 2008.

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

[21] N. Bansal et al., "Power Monitors A Framework for System-Level Power Estimation Using Heterogeneous Power Models," VLSID, pp. 579-585, 2005.

[22] S. Pasricha et al., "CAPPS: A Framework for Power-Performance Trade-Offs in Bus Matrix Based On-Chip Communication Architecture Synthesis", TVLSI, pp. 300-305, 2009.