Analysis and Evaluation of Greedy Thread Swapping Based Dynamic Power Management for MPSoC Platforms

Chirag Ravishankar[1], Sundaram Ananthanarayanan[1,2], Siddharth Garg[1], Andrew Kennings[1]
[1] University of Waterloo, Waterloo, ON Canada
[2] Anna University, Chennai, India
Corresponding Author E-mail: siddharth.garg@uwaterloo.ca

Abstract— Thread migration (TM) is a recently proposed dynamic power management technique for heterogeneous multi-processor system-on-chip (MPSoC) platforms that eliminates the area and power overheads incurred by fine-grained dynamic voltage and frequency scaling (DVFS) based power management. In this paper, we take the first step towards formally analyzing and experimentally evaluating the use of power-aware TM for parallel data streaming applications on MPSoC platforms. From an analysis perspective, we characterize the optimal mapping of threads to cores and prove the convergence properties of a complexity effective greedy thread swapping based TM algorithm to the globally optimal solution. The proposed techniques are evaluated on a 9-core FPGA based MPSoC prototype equipped with fully-functional TM and DVFS support, and running a parallelized video encoding benchmark based on the Motion Picture Experts Group (MPEG-2) standard. Our experimental results validate the proposed theoretical analysis, and show that the proposed TM algorithm provides within 8% of the DVFS performance under the same power budget, and assuming no overheads for DVFS. Assuming voltage regulator inefficiency of 80%, the proposed TM algorithm has 9% higher performance than DVFS, again under the same total power budget.

Keywords— Power management, Thread migration, DVFS, Multi-core, FPGA

I. Introduction

Highly parallel multi-processor system-on-chip (MPSoC) platforms are increasingly being used in embedded and mobile computing devices, where, besides performance, the system power dissipation is a first-class design constraint. Dynamic power management techniques aim to exploit the *spatial* and *temporal* variations in application characteristics to either save power within a performance budget, or maximize performance for a specified power level. To provide effective dynamic power management capabilities, a number of researchers [16], [9], [11] have proposed the use of fine-grained dynamic voltage and frequency scaling (DVFS) for MPSoC platforms. At the finest granularity, DVFS allows *each* core to independently adapt its voltage and frequency at run-time to suit the requirements of the thread executing on that core. For example, performance critical threads can be run at high voltage and frequency levels to maintain performance, while non-critical threads can be run at lower voltage and frequency to save power.

However, while DVFS can be an effective dynamic power management technique, it comes with associated overheads in terms of: (i) the area and power dissipation of the additional circuitry needed to support DVFS, (ii) the inefficiency of the voltage regulators required to dynamically scale the supply voltage at run-time, and (iii) constraints on the voltage regulator switching speed which limits how frequently the supply voltage can be scaled. The area and power overheads scale with the number of cores in the MPSoC platform, if fine-grained DVFS support is provided.

As an alternative, thread migration (TM), originally proposed as a mechanism to balance the thermal profile of multi-core chips [10], [7], [4], has been proposed as a low-overhead dynamic power management solution for *heterogeneous* multi-core platforms [17], [6]. The key idea is to *statically* set the voltage and frequency value of each core at design time (resulting in a heterogeneous system where different cores operate at different power and performance levels), and then, to *dynamically* migrate threads to appropriate cores at run-time. This is in contrast to DVFS, where the thread mapping remains fixed, but the voltage and frequency of each core is dynamically updated based on the performance requirements of the thread running on it. Rangan et al. [17] have evaluated, using software simulations, the use of thread migration based dynamic power management for a multi-core system running *multi-programmed* workloads, i.e., where each thread represents an independent application. More recently, Dighe et al. [6] have implemented and evaluated the benefits of thread migration on an 80-core hardware prototype using a simple synthetic application that consists of only three independent threads and a hard-coded migration sequence. While these works illustrate the potential of TM as an alternate or complementary power management solution to DVFS, the efficacy of power-aware TM for parallel, multi-threaded applications (as opposed to multi-programmed or synthetic applications used by prior work) has not been analyzed and experimentally evaluated before.

In this paper, we take the first step towards *formally analyzing the optimality and convergence properties of power-aware TM for parallel applications running on heterogeneous MPSoC platforms.* In particular, we focus on streaming applications [18] for which the execution semantics are specified as homogeneous synchronous data flow graphs. This model encompasses a wide range of applications commonly used in embedded and mobile platforms including video processing, base-band communications and digital signal processing. Based on this analysis, we propose a complexity-effective TM algorithm — greedy thread swapping —- and evaluate its performance on a 9-core FPGA prototype enabled with per-core static and dynamic frequency scaling, and full support for run-time thread swapping between any two cores. As a representative benchmark, we use a custom, parallel implementation of a video encoder based on the Motion Picture Experts Group (MPEG-2) standard that runs on eight of the nine cores on the FPGA platform, while the ninth core is used to implement the TM (or DVFS) algorithm and to orchestrate the thread migration. Our experimental results illustrate that TM is indeed a promising, low overhead alternative to fine-grained DVFS for MPSoC platforms — in particular, we observe that compared to ideal DVFS with no overheads, the performance of TM is only 8% lower than DVFS within the same power budget. Accounting for the power overheads of the voltage regulators required to implement DVFS, the performance of TM is actually 9% higher than the DVFS performance, again within the same power budget. Compared to a static mapping of threads on to a heterogeneous MPSoC platform, TM results in up to 3× higher throughput.

II. Related Work

Dynamic power management for MPSoC platforms has primarily been addressed in the context of fine-grained or clustered DVFS, for which a number of algorithms have been proposed in literature based on non-linear optimization [14], [15] and control theory [16], [9], [1], among other techniques. While an exhaustive survey of work in this area is outside the scope of this paper, we note that these DVFS algorithms can *not* be simply extended for the case of power-aware TM. This is because DVFS is fundamentally an *assignment* problem, i.e., the goal is to assign a voltage and frequency value to each thread to meet the desired objective. On the other hand, power-aware TM is an instance of a bipartite *matching* problem, i.e., the goal is to match each thread to an *available* set of statically determined voltage and frequency values.

TM has been proposed as a low-overhead solution for thermal balancing in multi-core systems [12], [7], [10], [4] and more recently, to minimize the inter-thread communication costs [13]. Compared to these related works, this paper has a different objective, i.e., dynamic power management for heterogeneous MPSoC platforms, where each core can have different statically set voltage and frequency values.

The recent work by Rangan et al. [17] and Dighe et al. [6] is the most closely related to the work presented in this paper. Compared to these papers, this work makes the following novel contributions:

- We address power-aware TM for truly parallel applications as opposed to the multi-programmed and synthetic workloads that are used by [17] and [6]. We present our results on an 8-thread parallel implementation of a video encoding algorithm based on the MPEG-2 standard.
- We *formally* characterize the optimal mapping of a parallel application — the mapping that maximizes application throughput within a power budget — on a heterogeneous MPSoC platform and prove that a complexity-effective *greedy thread swapping* algorithm is guaranteed to converge to the optimal solution in a bounded number of steps.
- We experimentally evaluate our proposed solutions on an FPGA prototype of 9-core MPSoC running the VE benchmark. The prototype fully supports thread swapping between arbitrary pairs of cores and therefore, in contrast to simulation based studies [17], our results accurately reflect *any* performance overheads associated with TM that would be observed in a real system.

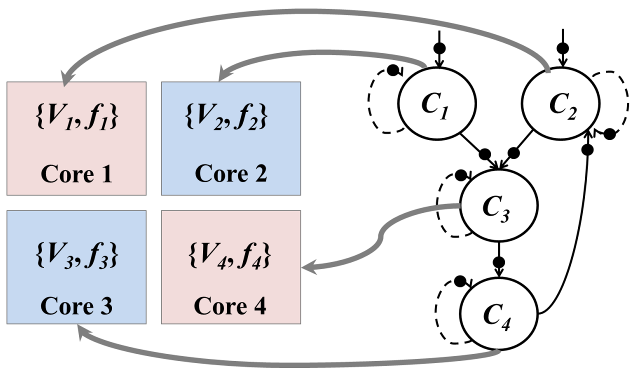

Fig. 1. MPSoC architecture and application models assumed in this paper.

III. Preliminaries and Assumptions

We now discuss the details of the MPSoC architecture and parallel application model that is assumed in this work. **MPSoC Architecture.** As shown in Figure 1, we assume a heterogeneous MPSoC architecture consisting of N cores, where the supply voltage of core i is given by V_i, its clock frequency by f_i, and the cycle time $T_i = \frac{1}{f_i}$. Also, we assume (as assumed by [17] and [6]) that the cores are architecturally identical, and that the only source of heterogeneity is that the cores operate at different voltage and frequency levels. Finally, the communication between cores occurs via dedicated point-to-point, mixed-clock FIFO queues between each pair of cores. The FIFOs are used to communicate both application data and for swapping architectural state during TM.

Application Model. In this paper, we model applications as homogeneous synchronous data-flow (HSDF) graphs [5], which is a commonly used model for a large number of streaming data applications. In particular, the application is modeled as a (possibly cyclic) graph, $G(V, E)$, where the vertices V represent computational tasks, and the edges represent the data-flow dependencies between tasks. We assume that the number of tasks is strictly less than or equal to the number of available cores on this MPSoC platform, i.e., $|V| \leq N$. Each time a task executes, it consumes one token of data and produces one token of data at each output edge. We assume that task i takes C_i clock cycles to process one token of data, and for now, assume also that C_i does not change with time.

The mapping function, M, maps application tasks to the cores in the MPSoC platform — $M(i) = j$ implies that task i is executing on core j. Given a mapping M, we can write the execution latency of task i, $L_i = C_i T_{M(i)} = \frac{C_i}{f_{M(i)}}$. Finally the steady-state *throughput* of a mapping M, $TP(M)$, can be written as [5]:

$$TP(M) = \min_{\forall Q \in G} \frac{|Q|}{\sum_{i \in Q} C_i T_{M(i)}}$$

where Q represents *cycles* in the HSDF graph G[1]. Note that besides data flow cycles in the graph G, for example the one from task $2 \to 3 \to 4 \to 2$ in Figure 1, each task also has a self-loop that models the fact that multiple instances of the task cannot execute simultaneously.

Note that in practice, the number of execution cycles for each task are data dependent and will change with time — the goal of the TM algorithm is to dynamically update the mapping of tasks to cores, M, so as to *maximize* the application throughput within a power budget. To this end, we begin by determining the optimal mapping M^* assuming that the number of execution cycles for each core is known, and then outline a complexity-effective greedy thread swapping algorithm that is guaranteed to converge to this optimal mapping in a finite number of steps.

IV. Power-aware Thread Migration

We begin by showing that for the case in which each task in the data-flow graph is mapped to an individual core, the application throughput is limited by self-loops, in other words, the worst-case throughput is limited by the thread that has the largest execution latency.

Lemma 1: The steady-state throughput of a HSDF graph G mapped onto an MPSoC platform using the mapping M can be written as $TP(M) = \frac{1}{\max_{i \in V}(C_i T_{M(i)})}$.

Proof: For any cycle Q in G not including self loops, we note that $\frac{\sum_{i \in Q} C_i T_{M(i)}}{|Q|} \leq max_{i \in Q}(C_i T_{M(i)})$. This shows that the cycles in G that are not self loops can not be throughput constraining. Therefore, the throughput constraining cycle *must* be one of the $|V| \leq N$ self loops in the graph, i.e., $TP(M) = \frac{1}{\max_{i \in V}(C_i T_{M(i)})}$. ■

Based on Lemma 1, we can now characterize an *optimal* mapping M^* that maximizes throughput for an application under a power budget. Note that the power budget is implicitly determined by the statically set voltage and frequency values of each core.

A. Globally Optimal Mapping

Theorem 1: Given a parallel application with N tasks specified as an HSDF graph, $G(V, E)$, (without loss of generality, assume that the number of execution cycles are ordered as follows: $C_1 \leq C_2 \leq \ldots \leq C_N$), and an MPSoC platform with N cores (without loss of generality, assume that the cycle times of the cores are ordered as follows: $T_1 \geq T_2 \geq \ldots \geq T_N$), the optimal mapping of threads to cores, M^*, is given by $M^*(i) = i$.

Proof: We will prove the result via induction on N, the number of threads and cores in the system. For $N = 2$, we can show that $max(C_1T_1, C_2T_2) \leq max(C_1T_2, C_2T_1)$. This is because $max(C_1T_2, C_2T_1) = C_2T_1$, and C_2T_1 is greater than both C_1T_1 and C_2T_2.

Assume the statement is true for P threads and P cores, where $P < N$. We show that, given this assumption, it is also true for a system with N threads and N cores.

Assume an arbitrary mapping, M, of N threads to N cores. Without loss of generality, assume that thread 1 runs on core i. Now, we define a new mapping M' such that:

$$M'(1) = i$$

$$M'(j) = j - 1 \quad \forall j \in [2, i]$$

$$M'(j) = j \quad \forall j \in [i + 1, N]$$

In the mapping M', thread 1 is mapped to core i which is also the case in mapping M, but note that the remaining $N - 1$ threads are *optimally* mapped on the remaining $N - 1$ cores, based on the induction assumption. For example, thread C_2, the least computationally expensive of the remaining $N - 1$ threads is mapped on core 1, the slowest of the remaining $N - 1$ cores, and so on. Therefore, $TM(M') \geq TM(M)$.

Now comparing mapping M^* to M', we can observe that in both cases, $M'(j) = M^*(j) = j$ ($\forall j \in [i + 1, N]$). However, based on the induction assumption, the remaining i threads are optimally mapped on the remaining i cores in the mapping M^*, but *not* in M'. As a result, $TM(M^*) \geq TM(M')$, and thus, $TM(M^*) \geq TM(M)$, where M is any arbitrary mapping of threads to cores. This completes the desired proof. ■

Theorem 1 proves that the intuitive mapping solution, i.e., mapping the most computationally expensive thread to the fastest core, the second most computationally expensive thread to the second fastest core (and so on), is indeed the globally optimal solution for the parallel data streaming application model that we study in this paper.

B. Greedy Thread Swapping

From a practical perspective, switching from an arbitrary initial mapping M to the optimal mapping M^* within one step might require multiple threads or, in the worst case, each thread to be migrated to a different core. We refer to this scenario as *global thread migration.* For example, consider the scenario in Figure 2, where Thread 2, Thread 3 and Thread 4 all participate in migration, and form a cyclic dependency chain. Such migrations can be complex to implement in practice — in fact, global thread migration is impractical for many reasons: (i) increased performance overheads because of the amount of network traffic required to perform the migration, and because a larger number of cores start with cold caches [3]; (ii) increased

[1]Note that we implicitly assume, as assumed by [5] and [8], that each incoming edge has an initial token.

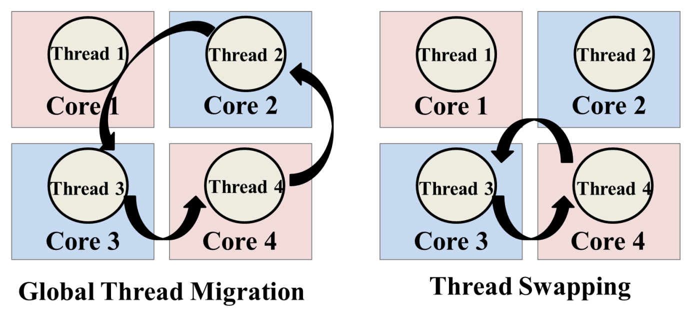

Fig. 2. Example illustrating both global thread migration (high overhead and implementation complexity) and complexity-effective thread swapping.

implementation complexity, for example, in our prototype implementation we observed a marked increase in implementation complexity moving from two-core thread migration (i.e., thread swapping) to global thread migration for three cores; and (iii) potential for deadlocks, as observed by [2].

Therefore, as a number of prior researchers have done, we focus on simple *thread swapping* where, during any migration interval, an arbitrary pair of cores in the MPSoC can swap threads. This is illustrated in Figure 2. Thread swapping addresses many of the performance overhead, implementation complexity and deadlock issues related to global thread migration, but also implies that it is no longer possible to switch from an arbitrary low performance mapping, M, to the optimal mapping, M^*, in a single control interval.

We now present a simple Greedy Thread Swapping (GTS) algorithm, and then prove, analytically, that, starting from a sub-optimal initial mapping M, GTS is guaranteed to converge to M^* in a finite number of steps. While similar algorithms have been proposed by prior researchers [17], this is, to the best of our knowledge, **the first formal proof of the convergence properties of this algorithm to the globally optimal solution**.

```
1:  Given $C = \{C_1, C_2, \ldots, C_N\}$
2:  Given $T = \{C_1, C_2, \ldots, C_N\}$
3:  Given a mapping $M : C \rightarrow T$
4:  $p = \arg\max_{i\in[1,N]}(C_i T_{M(i)})$
5:  $\Delta_{max} = 0$
6:  for $j = 1 \rightarrow N$ do
7:      $\Delta = C_p T_{M(p)} - \max(C_p T_{M(j)}, C_j T_{M(p)})$
8:      if $\Delta \geq \Delta_{max}$ then
9:          $s = j$
10:         $\Delta_{max} = \Delta$
11:     end if
12: end for
13: swap(p,s)
```

Algorithm 1: Greedy Thread Swapping (GTS)

We now prove that, starting from *any* initial mapping M, where $TP(M) < TP(M^*)$, GTS always converges to the mapping M^* in a finite number of steps, i.e., thread swaps. To prove this, it is sufficient to prove that, given any arbitrary mapping M, there *always* exists at least one swap that increases the application throughput — in other words, that the throughput as a function of the space of possible mappings has no local maxima.

Theorem 2: Given an arbitrary mapping M where $TP(M) < TP(M^*)$, there exists a new mapping M', where (i)$M'(p) = M(s)$, (ii) $M'(s) = M(p)$, and (iii) $M'(k) = M(k)$ ($s \neq p$ and $\forall k \neq s, p$), such that $TP(M') > TP(M)$.

Proof: We will again assume, without loss of generality, that $C_1 \leq C_2 \leq \ldots \leq C_N$ and that $T_1 \geq T_2 \geq \ldots \geq T_N$.

Let $p = \arg\max_{i\in[1,N]}(C_i T_{M(i)})$, i.e., p represents the throughput constraining thread in the mapping M. Since $TP(M) < TP(M^*)$, we know that $T_{M(p)} > T_{M^*(p)}$, i.e., $T_{M(p)} > T_p$. Thus, there must exist an $s \in [p+1, N]$ for which $M(s) < p$. In other words, there exists a thread s which is more computationally intensive than the throughput constraining thread p, and is mapped to a slower core compared to p. Thus, a new matching, M' obtained from swapping these threads will have a higher throughput, i.e., $T(M') > T(M)$. ■

Theorem 2 is an automatic proof of convergence to optimality, since (i) there exists an upper bound on throughput $TP(M^*)$; (ii) the space of possible mappings is finite; and (iii) there always exists a swap that increases throughput given a mapping that has sub-optimal throughput.

It is important to note that the proof is valid under the implicit assumption that the thread characteristics, i.e., the number of execution cycles remain constant. In reality, the thread characteristics are workload dependent and vary with time. However, all dynamic power management algorithms, including ours, are predicated on a notion of temporal locality, i.e., that the workload variations are correlated in time, or do not change "too quickly". In the context of GTS, the key question is if the workload varies faster than the number of iterations GTS takes to converge to optimality. In the experimental results section, we show that, in fact, for the workloads evaluated, GTS is able to effectively adapt to workload variations.

V. FPGA Based Evaluation Platform

System-level or cycle-accurate simulations of MPSoC are commonly used to evaluate the performance of dynamic power management algorithms, often at the expense of fidelity. In the TM context, for example, Brooks et al. [17] use cycle-accurate simulations of a clustered multi-core processor, but only estimate the overheads of TM by assuming a certain on-chip bandwidth for data transfer during thread migration. To ensure the highest fidelity in evaluating the performance of the proposed TM algorithms, we have developed a multi-core FPGA based platform consisting of nine embedded processors that communicate via a point-to-point FIFO based interconnect. In addition, the frequency of each core can be independently set, either statically or dynamically — the latter enables us to use the same platform to evaluate and contrast TM with fine-grained DVFS.

To enable thread swapping on the FPGA platform, one of the nine cores acts as a dedicated **monitor** core that orchestrates TM, while the other eight **user** cores execute application threads. The monitor core periodically raises an interrupt to signal thread migration events (the period between two thread migration events is called the control interval T_{cont}), upon which user cores call a custom interrupt routine that performs the actions required to swap threads. The overheads of TM come from two sources - (i) the private instruction and data caches of each cores are flushed during TM and (ii) for correctness, the data in the inter-core FIFOs needs to be migrated to the appropriate FIFOs *post* swap.

Modeling Power Consumption. The FPGA platform currently does *not* have support for measuring either full-chip or per-core power consumption. We therefore use a coarse-grained power model to estimate the power consumption of each core. In particular, the instantaneous power consumed by core i, $Power_i$, is computed as:

$$Power_i \propto V_i^2 f_i$$

Since the FPGA board supports only frequency scaling and not voltage scaling, the voltage, V_i, corresponding to a frequency level f_i is estimated using published data for the voltage frequency relationship of embedded cores in the single cloud chip SCC [11] platform. The exact values used are discussed in the experimental results section, but we note that the the lack of voltage scaling does *not* impact the fidelity of the performance measurements on the FPGA platform, since the performance of each core depends only on its operating frequency.

VI. Experimental Results

We now discuss our experimental results that evaluate the performance of the proposed greedy thread swapping algorithm (GTS) on the FPGA based MPSoC prototype running a parallel video encoding (VE) benchmark. We focus specifically on validating the convergence properties of GTS starting from arbitrary initial mappings, illustrating the benefits of dynamic thread migration compared to static mapping, and comparing the performance of GTS to fine-grained DVFS. We begin by discussing the details of the VE benchmark.

A. Video Encoding Benchmark

Figure 3 shows the data flow graph of the VE benchmark ($\approx$ 3000 lines of C++ code) that consists of a Source core that sends raw video frames to the remaining cores, seven parallel processing threads, and the Frame Buffer which is implemented using on-chip memory. In our implementation, the VE encodes two types of frames:- (i) **I-Frames** that do not make use of the motion estimator (ME) and are directly compressed; and (ii) **P-Frames** in which the ME block is used to determine the macro-blocks (MB) in a previously processed frame that are in the neighborhood of the current MB being processed. The ME is the primary

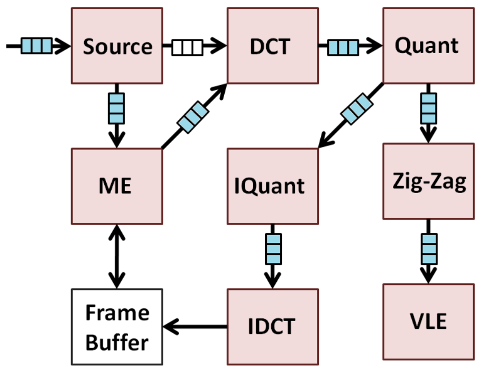

Fig. 3. Data flow graph of the VE benchmark.

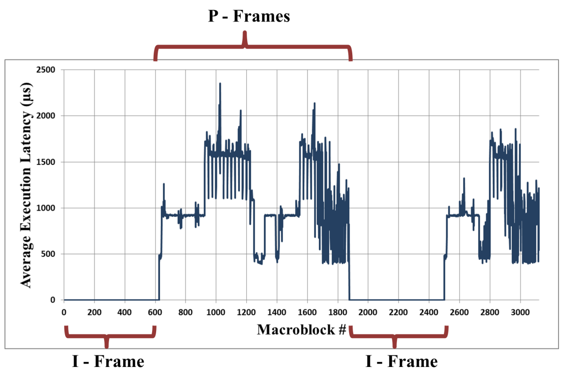

Fig. 4. Execution times for the ME thread.

source of workload variations for the VE benchmark, as shown in Figure 4. We can observe that while the execution time for ME can vary significantly for different MBs, the variations are temporally correlated, i.e., the execution time of a MB can typically be used as a good predictor for the execution time of the succeeding MBs. Both the DVFS and proposed GTS algorithms make use of this property to optimize performance.

B. FPGA Based MPSoC Prototype Details

TABLE I
Architectural details of the FPGA prototype

Combinational ALUTS	46041/182400 (25%)
Memory ALUTS	272/91200 (<1%)
DSP Blocks	36/128 (3%)
Available Clock Frequencies	{50, 37.5, 25, 12.5} MHz
Inter-core FIFO Size	509 Bytes
Nios II I-Cache	4 KBytes
Nios II D-Cache	2 KBytes

Our FPGA prototype is developed on an Altera Stratix IV GX FPGA board using embedded NIOS II cores. Table I provides a highlight of the synthesis results and important architectural details. Note that there are four frequency values available for static and dynamic frequency scaling that range from 12.5 MHz to 50 MHz. Since we can not measure per-core or full-chip power dissipation on the FPGA platform, we use published data for the voltage and frequency scaling behavior of embedded cores in the Intel SCC platform to determine how the voltage scales as the frequency scales by 4×, i.e., from 50 MHz to 12.5 MHz, and then used the coarse grained power model described in the previous section to determine the power consumption at every frequency level. This data is shown in Table II. As we can see, the power consumption for the 12.5MHz core is only 7.5% of the power consumption of a 50 MHz core.

TABLE II
VOLTAGE AND POWER VALUES (IN ARBITRARY UNITS (A.U.)) CORRESPONDING TO EACH FREQUENCY LEVEL.

Frequency (MHz)	Voltage (V)	Power (A.U.)
12.5	0.75 V	0.075
25	0.83 V	0.2
37.5	1.00 V	0.4
50	1.32 V	1.0

C. *Greedy Thread Swapping: Static Workload*

We begin by validating the convergence properties of the GTS based TM algorithm for *static* workloads, i.e., when the thread execution latency does not change with time. Since the ME thread is the only thread that has data dependent execution times, we ran the VE benchmark to process each frame as an I-frame, thus shutting off the ME core and eliminating workload variations. The execution time of each thread per MB is shown in Table III. With the ME shut off, the Quant thread is the most computationally expensive thread in our VE implementation.

TABLE III
AVERAGE EXECUTION TIME (IN CLOCK TICKS) TO PROCESS A MB FOR AN I-FRAME FOR EACH THREAD IN VE BENCHMARK.

DCT	Quant	Zig-Zag	VLE	I-Quant	I-DCT
20590	28290	5447	3796	3268	14506

With this benchmark, we ran an experiment on a heterogeneous MPSoC platform with three cores at 12.5 MHz, two at 25 MHz, one at 37.5 MHz and two at 50 MHz, with the Source always running at 50 MHz. Figure 5 shows the throughput increasing with every control interval for three different sub-optimal initial mappings of threads to cores. It can be observed that in all three cases, the throughput converges to its maximum value after a finite number of control intervals, as predicted by the theoretical analysis of the GTS algorithm. To illustrate the functioning of the GTS algorithm, Figure 6 shows the first initial mapping and the sequence of swaps made by the GTS algorithm to reach the optimal mapping, after which no swaps are made.

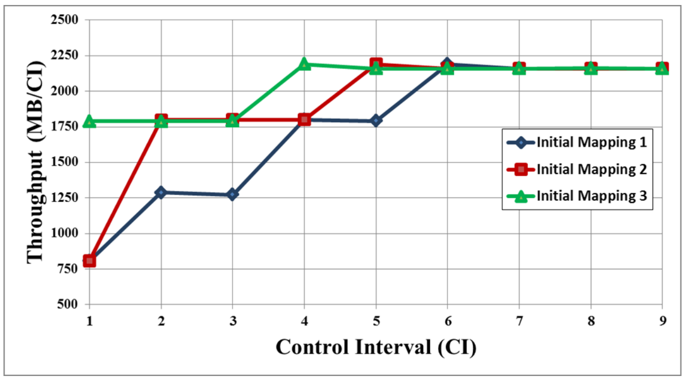

Fig. 5. Throughput (in terms of macro-blocks processed) at each control interval for different initial mappings using the GTS algorithm. We can see, that as predicted by the theoretical results, the optimal mapping that maximizes steady-state throughput is reached in a finite number of steps.

Control Interval	**Core 1** 50 MHz	**Core 2** 12.5 MHz	**Core 3** 12.5 MHz	**Core 4** 12.5 MHz	**Core 5** 25 MHz	**Core 6** 25 MHz	**Core 7** 37.5 MHz	**Core 8** 50 MHz
1	Source	DCT	Quant	Zig-Zag	VLE	I-Quant	ME	I-DCT
2	Source	DCT	**ME**	Zig-Zag	VLE	I-Quant	**Quant**	I-DCT
3	Source	DCT	ME	Zig-Zag	VLE	I-Quant	**I-DCT**	**Quant**
4	Source	**I-DCT**	ME	Zig-Zag	VLE	I-Quant	**DCT**	Quant
5	Source	**I-Quant**	ME	Zig-Zag	VLE	**I-DCT**	DCT	Quant
6	Source	I-Quant	ME	**VLE**	**Zig-Zag**	I-DCT	DCT	Quant

Fig. 6. Sequence of swaps to reach maximum throughput for Initial Mapping 1. Shaded boxes represent the threads that swap in the given control interval.

D. *Greedy Thread Swapping: Time Varying Workload*

To evaluate the performance of GTS for realistic, time-varying workloads we ran the VE algorithm (this time in regular mode with both P-frames and I-frames) on the heterogeneous MPSoC configuration described in the previous section. Again, the Source is always statically set to run at 50 MHz. To understand how GTS works for this case, Figure 7 plots the location of each thread in each control interval during program execution. The application is started with a "bad" initial mapping of threads to cores. We notice three distinct phases during application execution — in the start-up phase, a number of swaps are made to switch to a better thread to core mapping. During P-frame processing, the ME and Quant benchmarks run at the two highest frequencies of 50 MHz and 37.5 MHz and there are frequent swaps between these two due to variations in the ME execution time. When I-frames are processed, the ME execution time drops to zero and the GTS algorithm quickly swaps ME out to the lowest frequency (12.5 MHz) to boost performance. Compared to executing with the initial mapping and no thread migration, the GTS based TM algorithm provides a 3.03× increase in throughput for the same power budget. Compared to a high power design where all cores run at $50MHz$ the heterogeneous MPSoC with TM has a 63% lower power dissipation with only 33% reduction in throughput.

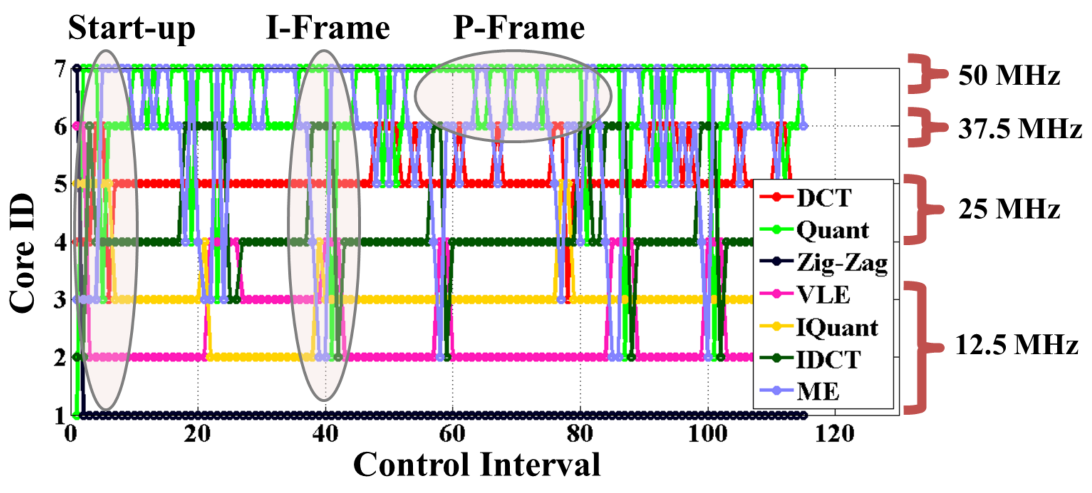

Fig. 7. Mapping of threads to cores in every control interval. The Source always runs at 50 MHz and is not shown here for clarity.

E. Greedy Thread Swapping Vs. DVFS

Finally, we compare the performance of GTS to DVFS for the same power budget. We implemented two versions of the DVFS algorithm — (1) Optimal DVFS in which, in every control interval, we exhaustively search all frequency combinations within the power budget and use the frequency assignment that maximizes performance, and (2) DVFS-TM where the frequency assigned to each thread is the same as that which would be assigned by the GTS based TM algorithm (although we note that GTS migrates threads instead of assigning frequencies). The former represents the best performance that can be achieved using DVFS, assuming 100% efficient voltage regulators. The latter, DVFS-TM provides insight into the performance lost by TM due to the overheads of swapping the threads and starting with cold caches after a TM event.

Figure 8 compares the throughput obtained (in Frames processed per second) for all three techniques for different control interval lengths (in clock ticks). Smaller control interval lengths imply that the thread migration events for GTS, or voltage/frequency updates for DVFS occur more frequently. A number of observations can be made from the figure:

- The throughput of GTS based TM is low for both large control intervals (no adaptation to workload variations) and small control intervals (increased overheads of repeated thread swapping). Compared to Optimal DVFS, the highest throughput for GTS is only 8% lower than the highest throughput for DVFS within the same power budget. It is important to note that we do not account for *any* overheads related to the additional circuitry required for DVFS or the voltage regulator inefficiencies, so the DVFS results are *optimistic*.
- Comparing DVFS-TM and GTS, we can infer that much of the performance loss compared to Optimal DVFS results from the run-time overhead of performing thread swaps. Without these overheads, the throughput of GTS based TM would be only 4.5% lower than the optimal DVFS throughput.
- While TM can be performed as frequently as desired (although the performance drops if it is performed too frequently), the length of the DVFS control interval is limited in practice by the switching speed of the voltage regulator. For this benchmark, our results suggest that the optimal length of a control interval is between 1 million and 5 million clock ticks, which, for a real system running at 1 GHz (as an example) would correspond to a control interval of between 1 ms and 5 ms.

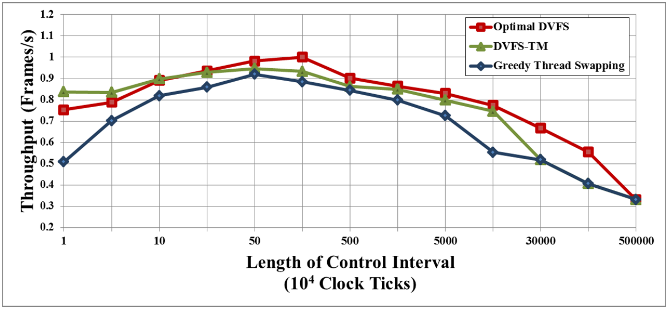

Fig. 8. Throughput of GTS compared to two DVFS algorithms for different control interval lengths.

Finally, we note that if the voltage regulator inefficiencies are accounted for, the power budget available to the cores for the system equipped with DVFS would reduce. Assuming a voltage regulator with 80% efficiency, the **throughput of the GTS based TM is 9% better than DVFS** for the same total power budget. For 90% regulator efficiency, the performance of DVFS and GTS based TM are within 2% of each other.

VII. Conclusions and Future Work

In this paper, we have taken the first step towards formally analyzing the optimality and convergence properties of thread migration based dynamic power management for heterogeneous MPSoC platforms running parallel data streaming applications. To this end, we propose a complexity-effective greedy thread swapping based thread migration algorithm, and prove theoretically that it converges to the optimal mapping of threads to cores in a finite number steps. We have evaluated the proposed techniques on a fully functional FPGA based MPSoC prototype with support for static and dynamic frequency assignment, and interrupt based thread swapping functionality. As a representative benchmark, we use a parallelized implementation of a video encoding application based on the MPEG-2 standard. Our experimental results not only confirm our theoretical analysis, but demonstrate that GTS based TM is very competitive with an optimal DVFS implementation (assuming no overheads for DVFS) and provide a maximum throughput that is within 8% of the throughput provided by Optimal DVFS for the same power budget. If voltage regulator inefficiency is accounted for, GTS based TM is up to 9% better than DVFS, again within the same power budget.

There are a number of avenues for future work. We would like to extend our analysis for more additional models of parallelism besides data streaming applications. Additionally, we would like to explore what impact having a network-on-chip (NoC) like interconnect, instead of point-to-point communication, would have on the performance of TM.

References

[1] A. Alimonda, S. Carta, A. Acquaviva, A. Pisano, and L. Benini. A feedback-based approach to dvfs in data-flow applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(11):1691–1704, 2009.

[2] M.H. Cho, K.S. Shim, M. Lis, O. Khan, and S. Devadas. Deadlock-free fine-grained thread migration. In *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, pages 33–40. IEEE.

[3] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance implications of single thread migration on a chip multi-core. *ACM SIGARCH Computer Architecture News*, 33(4):80–91, 2005.

[4] A.K. Coskun, T.S. Rosing, and K. Whisnant. Temperature aware task scheduling in mpsocs. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1659–1664. EDA Consortium, 2007.

[5] A. Dasdan and R.K. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(10):889–899, 1998.

[6] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, et al. Within-die variation-aware dynamic-voltage-frequency scaling core mapping and thread hopping for an 80-core processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 174–175. IEEE, 2010.

[7] T. Ebi, M.A. Al Faruque, and J. Henkel. Tape: thermal-aware agent-based power economy for multi/many-core architectures. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 302–309. ACM, 2009.

[8] S. Garg and D. Marculescu. System-level throughput analysis for process variation aware multiple voltage-frequency island designs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(4):59, 2008.

[9] S. Garg, D. Marculescu, and R. Marculescu. Custom feedback control: enabling truly scalable on-chip power management for mpsocs. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pages 425–430. ACM, 2010.

[10] E. Kursun, C.Y. Cher, A. Buyuktosunoglu, and P. Bose. Investigating the effects of task scheduling on thermal behavior. In *Third Workshop on Temperature-Aware Computer Systems (TACS06)*. Citeseer, 2006.

[11] T. Mattson, R. Van der Wijngaart, et al. The intel 48-core single-chip cloud computer (scc) processor: Programmers view. In *Int. Conf. High Performance Computing*, 2010.

[12] P. Michaud, A. Seznec, D. Fetis, Y. Sazeides, and T. Constantinou. A study of thread migration in temperature-constrained multicores. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(2):9–es, 2007.

[13] M. Misler and N. Enright Jerger. Moths: mobile threads for on-chip networks. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 541–542. ACM, 2010.

[14] S. Murali, A. Mutapcic, D. Atienza, R. Gupta, S. Boyd, and G. De Micheli. Temperature-aware processor frequency assignment for mpsocs using convex optimization. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*, pages 111–116. IEEE, 2007.

[15] K. Niyogi and D. Marculescu. Speed and voltage selection for gals systems based on voltage/frequency islands. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 292–297. ACM, 2005.

[16] U.Y. Ogras, R. Marculescu, and D. Marculescu. Variation-adaptive feedback control for networks-on-chip with multiple clock domains. In *Proceedings of the 45th annual Design Automation Conference*, pages 614–619. ACM, 2008.

[17] K.K. Rangan, G.Y. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 302–313. ACM, 2009.

[18] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 49–84. Springer, 2002.