Pattern-Aware Dynamic Thread Mapping Mechanisms for Asymmetric Manycore Architectures

Jason A. Poovey, Michael C. Rosier, Thomas M. Conte -- Georgia Institute of Technology

Abstract

Recent factors in the architecture community such as the power wall and on-chip complexity have caused a shift to manycore architectures and multi-threaded workloads. An emerging architecture for general purpose processing is an asymmetric chip layout composed of cores with varying levels of complexity and power. This work considers thread behavior that results from parallel programs and proposes four new thread mapping schemes for this asymmetric architecture. These schemes leverage the behavior of common programming patterns used in parallel programming to target typical thread behavior. The patterns targeted include the pipeline parallelism, divide and conquer, and recursive data patterns. The proposed predictors out-perform base asymmetry aware schemes by at most 40% and existing dynamic schemes by as much as 23% for the PARSEC and SSCA2 benchmarks. Additionally, this work demonstrates the relationship between thread heterogeneity and the parallel pattern used to create the workload, thus creating a link between programmer and architect that can be leveraged to create more intelligent thread mapping schemes.

1. Introduction

During the last decade factors such as the power wall and on-chip complexity have resulted in the shift towards manycore architectures and multi-threaded workloads. Rather than invest transistors and chip area in a single, complex out-of-order core that may prove to exceed the power envelope, the trend is to include more and more simpler cores on a single chip. This has led to several visions for the "architecture of the future." Commercial desktop processors have trended towards a few large, powerful "heavyweight cores" [1, 2], whereas GPU developers and the scientific community have focused on manycore chips consisting of 1000s of simple "lightweight cores" on a single chip [3-5]. An alternative view of our multi-threaded future is that neither the heavyweight- or lightweight-core designs will become the new "general purpose." Multicore microprocessors will instead have a combination of both types of cores. Prior work has demonstrated that with only a few heavyweight cores supplemented by several simpler, lightweight cores, it is possible to achieve performance comparable to many heavyweight cores [6-8]. The key to maximizing the gain of these heavyweight cores is to intelligently map a program's threads to the appropriate core types.

Thread mapping for operating systems and architecture has been a heavily investigated field [9-16]. Prior work has focused primarily on mapping compute-intensive threads to the heavyweight cores and communication-intensive threads to the smaller cores [14]. The compute-intensive threads are often

termed the *critical thread*. Several parallel programming styles result in applications that have imbalance in execution time among executing threads. For example, a common pattern for parallelizing legacy serial code is to create a parallel pipeline [17]. Creation of these pipeline stages results in imbalance between the execution requirements of each stage [18]. When this occurs, the thread with the highest execution demands dictates the overall performance, thus creating a *critical thread*.

When writing parallel applications, programmers attempt to perform load balancing statically or algorithmically. However, without a detailed knowledge of the architecture and compiler, it is difficult to predict load balancing prior to run-time. Many prior approaches have suggested a profiling based approach to determine the compute-intensive threads [10, 15, 19], while others measure various architecture statistics to predict thread criticality [13-16]. In this work we propose four new hardware-assisted thread mapping schemes that leverage runtime statistics: *maximum average dependence depth mapping* (MAX_DEP), *maximum average dependence length mapping* (MAX_LEN), *child-parent aware mapping* (CPAM), and *static pipeline aware mapping* (SPAM). We compare our approaches to versions of recently published mapping schemes such as *thread age using instruction count* (IC) [15], and *L1/L2 miss rate criticality prediction* (L2) [14]. We find that our new mapping schemes outperform prior approaches particularly well for pipeline parallel, divide and conquer, imbalanced geometrically decomposed, and recursive data benchmarks. In imbalanced workloads, we see a performance gain over basic asymmetry aware mapping as high as 40% for real benchmarks and as large as 23% for patter-centric microbenchmarks.

We also investigate the relationship between the parallel pattern used to write the benchmark and its thread behavior. By determining a link between parallel algorithm classes and thread behavior, programmers are better able to understand which load balancing schemes are optimal for their program. This enables a new level of exposure to the programmer and/or compiler to judiciously manage which thread mapping policy to enable prior to execution.

2. Motivation

Due to increasing core counts, a renewed focus has emerged for parallel programming to provide performance scaling. For many years the scientific community has leveraged thread parallelism to

perform massive calculations [20] and large simulations [21]. Often, these problems were either embarrassingly parallel or easily parallelizable through regular data structures. With the renewed focus on multi-threaded programming, the application space has grown dramatically and more irregular programs and application characteristics have emerged [22]. Asymmetric architectures are an ideal architecture for such workloads due to the inherent task imbalance that occurs between threads.

In order to understand this task imbalance and how to effectively counter it, one must first understand the reason for its existence. Despite the variety of parallel applications, programmers typically approach parallelization using only a small set of distinct patterns of design. Although sometimes a complex program may use a mixture of patterns, the understanding of the included patterns leads to an understanding of a unique set of thread characteristics.

2.1. Parallel Patterns and Thread Behavior

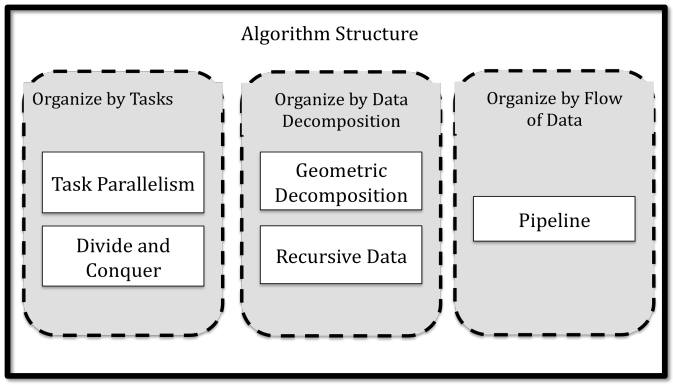

Figure 1 - Parallel Programming Patterns [23] showing the conceptual organization based on parallelization scheme.

Several efforts have been made to standardize parallel programming patterns [4, 17, 23]. These efforts codify the standards and characteristics in a manner similar to the programming patterns used in the software engineering community for object-oriented programming. What these standards reveal are five main patterns, each with unique architectural characteristics to exploit. The five parallel patterns are originally defined in [23] and are shown in Figure 1 (note we have combined event coordination and pipeline patterns, since the definition of event coordination in [23] reduces to a generalized pipeline as defined by Patel in Davidson in [24]).

As shown in Figure 1 the parallel programming patterns are grouped based on the type of conceptual parallelization performed. When the problem consists of a group of independent tasks or task groups to be run in parallel, the parallel pattern employed is ***task parallelism***. This class of problems has also been generally referred to as *embarrassingly parallel*. When there is a problem task that naturally subdivides into several smaller tasks that can be done in parallel, then the ***divide and conquer*** pattern is applied. Both of these parallel patterns involve the organization of tasks, and of the two divide and conquer has the greater potential for heterogeneous behavior.

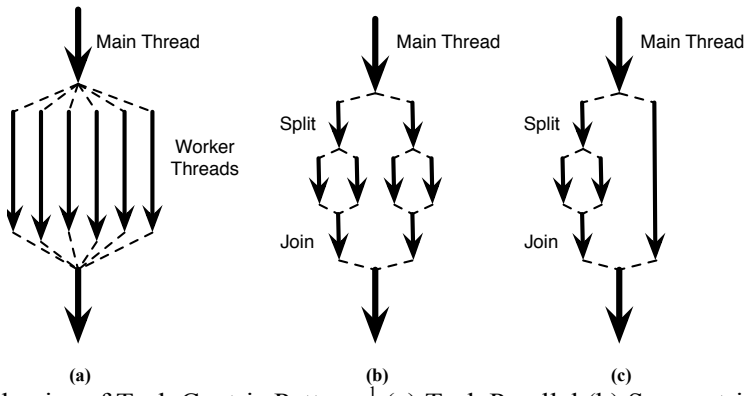

Figure 2 - Thread Behavior of Task Centric Patterns[1] (a) Task Parallel (b) Symmetric Divide and Conquer (c) Asymmetric Divide and Conquer

Task parallel workloads are generally written in a SIMD or SPMD program style where tasks are very balanced and have low communication overhead. Divide and conquer splits tasks until a work "threshold" is met and the subtask works on a subset of the data serially. When the divide and conquer is performed symmetrically, the opportunity for load balancing through task scheduling is low; when the division is more asymmetric, task scheduling presents an opportunity for improvements. For example, if a parallel quicksort algorithm is divided symmetrically, then all threads will continue to split their subarrays in half until some fixed threshold resulting in a uniform distribution of work. However, an asymmetric implementation would allow some threads to cease the recursive split with larger subarrays than others. Figure 2(a),(b),(c) illustrate the typical task behaviors of task parallel and divide and conquer patterns.

[1] Dashes represent communication between threads

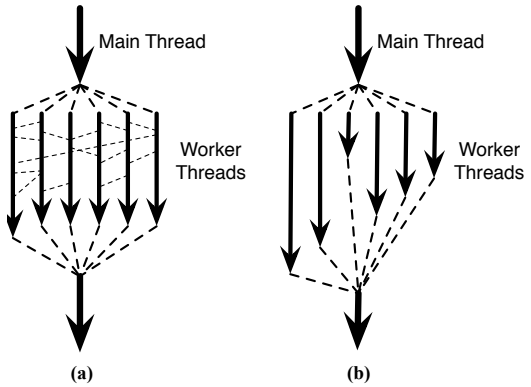

Figure 3 - Thread Behavior of Data Centric Patterns (a) Geometric Decomposition (b) Recursive Data

Many parallel problems are solved through the decomposition of data by creating threads to work on the data in parallel. The two standard patterns for data parallelization are ***geometric decomposition*** and ***recursive data***. The geometric decomposition pattern operates on data in a regular structure, such as an array, that is split into sub-structures operated on in parallel. This pattern is typically characterized by sharing between threads, particularly threads with neighboring data. Task heterogeneity is limited in geometrically decomposed workloads. Typically there is only one thread (usually the main thread) that exhibits heterogeneity as it maintains global data, bookkeeping, and thread management. If the data is not in a regular structure, but rather a structure such as a graph, data decomposition parallelization is done via the recursive data pattern. This pattern creates parallelism by doing redundant work to decrease communication between threads. For example, an algorithm to find every node's root requires a full graph traversal. A recursive data approach would create a thread for every node in the graph and perform a graph-climbing algorithm independently for each node. This causes some nodes' depths to be calculated more than once, but has performance gains due to the enhanced parallelism. Figure 3 (a) and (b) illustrates the typical task behaviors of the geometric decomposition and recursive data patterns.

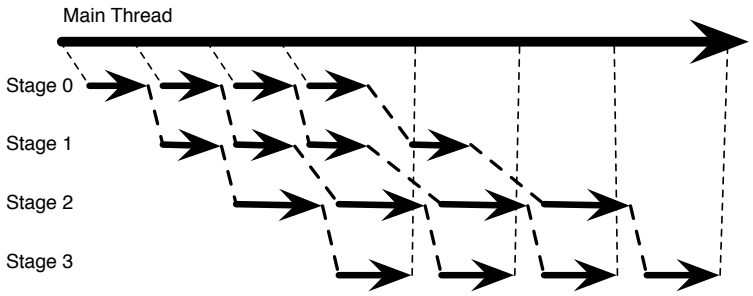

Figure 4 - Thread Behavior of the Pipeline Programming Pattern

As programmers continue to shift legacy sequential code to a parallel domain, an increasingly common parallel pattern used is the ***pipeline pattern*** [18]. This pattern is performed by taking a flow of data through tasks and splitting it into pipeline stages. The parallelism is achieved by keeping all stages full with data such that each stage can operate simultaneously. However, balancing the computational and communication requirements of each pipeline stage is very difficult. Moreover, not all pipeline parallel workloads are completely feed-forward pipelines. Simulations, such as discrete event simulations, leverage the pipeline pattern but with more complex interactions between stages. Therefore, pipeline programs typically exhibit a high degree of thread imbalance that can be exploited through thread remapping. Figure 4 illustrates the typical task behavior of the pipeline programming pattern.

Table 1 - Microbenchmark descriptions. Each benchmark represents unique thread behaviors exhibited by parallel design patterns.

Benchmark	Description
Divide and Conquer	Symmetric divide and conquer algorithm. Splits a size N array into subarrays, and finds the sum of the elements. Final output is the total sum
Divide and Conquer 2	Asymmetric Quicksort algorithm. Divides until a threshold. Two threads have a higher threshold. Once split insertion sort is used serially.
Geometric	2d array decomposed program. Neighboring array chunks communicate to calculate average of neighbors
Geometric 2	Alternative implementation of above
Pipeline	11 stage pipeline with imbalanced stages. Data is passed between stages using shared memory queues
Recursive	Graph traversal to find root nodes of all nodes. A thread is created per node.
Task Parallel	2d array split into chunks with no communication between nodes.

As the above discussion on patterns illustrates, thread heterogeneity is an important problem for current and emerging parallel workloads. We found that three out of five of the parallel programming patterns typically result in load imbalance that could be exploited by asymmetric cores. To show this pattern-based heterogeneity quantitatively, a set of experiments to measure thread imbalance was conducted. First, a set of microbenchmarks was created to represent each pattern. A summary of these microbenchmarks is given in Table 1. Two divide and conquer style benchmarks are included to provide a mix of symmetric and asymmetric behavior. In order to vary the mix of communication and

computation ratios, two geometric decomposition benchmarks were created with varying working set sizes and sharing volume.

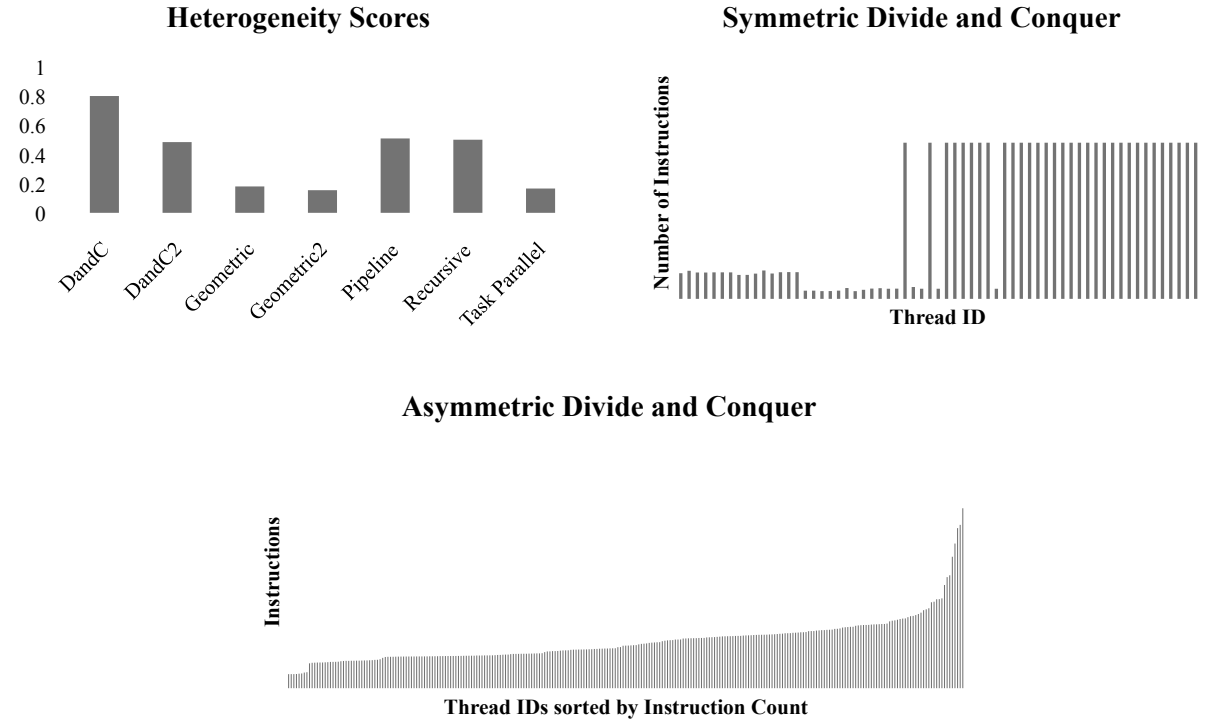

Figure 5 - Pattern Heterogeneity – top left: Heterogeneity Scores – top right: Symmetric Divide and Conquer thread behavior – bottom: Asymmetric Divide and Conquer thread behavior

We investigated the number of instructions per thread for each parallel pattern to show thread heterogeneity. Heterogeneity is quantized as the ratio of the standard deviation of the threads' instruction counts / the average of the instruction counts. Therefore a lower score indicates less heterogeneity, and vice versa. As Figure 5 illustrates, the patterns with the most heterogeneity are pipeline, recursive, and divide and conquer. Both symmetric and asymmetric divide and conquer exhibit heterogeneity, but as Figure 5 also shows, the heterogeneity in the symmetric divide and conquer is between levels of thread groups. Asymmetric divide and conquer exhibits no grouping of threads with similar instruction behavior.

Table 2 - Dependence Depth and Length Mapping Algorithms. "OOO_PROC" is a heavyweight (i.e., out-of-order) core

Dependence Length (MAX_LEN):	**Dependence Depth (MAX_DEP):**
On Fetch: (1) Update the dependence graph with the new instruction (2) Update length of all paths in the dependence graph	*On Fetch:* (1) Update the dependence graph with the new instruction utilizing the latest use times of the sources and destination and last completion times of the destination (2) Collect dependence depth of instruction based on the anti-, output-, and true- dependences (3) Use predicted latency values for ALU ops, and if a predicted L1 hit set latency to 3 cycles, otherwise latency is 100 cycles
On Schedule: (1) Collect average dependence lengths from all threads (2) Select the two threads with the highest average dependence length to map to the OOO_PROC (3) If threads are not already on OOO_PROC, remove any current running threads for the core and replace with the new critical threads (4) Reset length counters	*On Schedule:* (1) Collect average dependence depths from all threads (2) Select two threads with the highest average dependence depth to map to the OOO_PROC (3) If threads are not already on OOO_PROC, remove any current running threads from the core and replace with new critical threads (4) Reset depth counter

2.2. Pattern-centric thread prediction

Exploitation of each of the parallel patterns' heterogeneity is a unique and challenging problem. Any heterogeneity predictor must be balanced to consider that many workloads leverage more than a single parallel pattern. The first classes of predictors proposed in this work are ***dependence depth*** (MAX_DEP) and ***dependence length*** (MAX_LEN). These predictors are intended to target all three heterogeneous patterns, but primarily pipeline and recursive data. When there exists an inherent load imbalance, it is caused by one of two factors: varying thread length, varying thread complexity, or a combination of both. Thread length variance has been previously studied using instruction counts and thread age prediction [15]. Our predictors instead measure varying thread complexity through dependence chain analysis.

The dependence length algorithm (MAX_LEN) creates a data dependence graph at run-time as instructions are fetched from threads. It is undesirable for the dependence length to depend upon the type of core currently running the thread, therefore the weights of all dependence edges are set to 1 cycle. Table 2 summarizes the steps of the MAX_LEN algorithm. The dependence information is reset every

100ms during the standard thread scheduling interval. At every scheduling interval, the lengths of all dependence paths are averaged for each thread. The two threads with the highest average dependence length are predicted to be critical and are selected to run on the out of order cores.

The dependence depth algorithm (MAX_DEP) targets the same goal as dependence length, but instead of assuming all instructions have a dependence edge of one cycle, latency values are estimated using the predicted execution time. These latencies are fixed based on the opcode for non-memory instructions. Memory instructions peek into the L1 cache's tag store to predict if a hit will occur. If a hit is likely, the memory instruction is considered *short latency* and is set to 3 cycles; otherwise it is a *long latency* load and is set to 100 cycles to approximate a canonical miss penalty. The algorithm for dependence depth is summarized in Table 2.

Table 3 - SPAM and CPAM Mapping Algorithms

Static Pipeline Aware Mapping (SPAM):	**Child-Parent Aware Mapping (CPAM):**
On Fetch: (1) Update dependence graphs using MAX_DEP algorithm *On Thread Resume:* (1) If resuming thread is a *critical thread* (based on data from last scheduling interval) schedule on an OOO_PROC (2) If all OOO_PROCs are occupied, evict a non-critical thread from the core	*On Thread Spawn:* (1) Mark the child as critical and move to an OOO_PROC if not already occupied by critical threads (2) Mark the parent as non-critical (3) If parent was on OOO_PROC it gives up its core for its child *On Thread Exit:* (1) If a parent exists, mark as critical. (2) If child was on an OOO_PROC, give up core for parent

The stability of pipeline characteristics led us to develop the ***static pipeline aware mapping algorithm*** (SPAM). SPAM leverages the dependence depth algorithm from above, but does not reset the depth score at the end of a scheduling interval. SPAM is motivated by the behavior of pipeline benchmarks, which contain many suspend and resume events. These events are largely due to lock contention for the queues between stages. This then results in shorter running threads, and less opportunity for migration. Additionally, pipelines with large working sets have a higher cache migration penalty. Therefore it is important to map critical threads to heavyweight cores on thread-resume since

mid-thread migration will be costly and will not be mitigated by a long uninterrupted thread. The SPAM algorithm is summarized in Table 3.

SPAM utilizes the most recent criticality predictions to predict the critical threads. When a thread resumes, if it is one of the critical threads it will evict a non-critical thread from the OOO_PROC if needed and immediately get scheduled on the OOO_PROC. In order for SPAM to work properly the parallel pipeline must have stages that repeat several times. While the pipeline fills, the depth predictor will slowly learn which threads are critical. After the pipeline fills, the critical thread will remain largely static, and thus it will experience a consistent speedup throughout execution. For pipeline benchmarks with many shorter stages, SPAM provides better performance potential than MAX_LEN or MAX_DEP, because it mitigates costly mid-thread migrations.

The final proposed mapping algorithm is ***child-parent aware mapping*** (CPAM). This scheduling algorithm targets thread scheduling on thread spawn and exit. CPAM is motivated by the divide and conquer pattern in which threads recursively spawn sub-threads until the problem is split into a small enough unit to be operated on serially. In this pattern, it is true that when a parent spawns a child, the child becomes more critical than the parent. This is because the parent waits for the child to return with the subsolution, while the child performs the critical work. The CPAM algorithm is described in Table 3. CPAM gives the child thread preference to the out-of-order core. If the parent thread is currently scheduled on an out of order core, it will give up its core for the child. Once subsolutions are complete, the child will return its solution to the parent. The parent becomes the critical thread as it now has data on which to operate. Therefore, once the child exits, it gives its parent a preference for an out-of-order core. If the child is currently on an out-of-order core, it additionally gives up its core for the parent.

CPAM also complements the task-stealing algorithm leveraged for the dynamic predictors in the following fashion; when a thread suspends on an OOO_PROC, the task stealer attempts to find a thread to replace it. The task stealer first looks for cores with a preference for OOO_PROC. Because the CPAM algorithm will give preference to the children, as threads exit, other children will be available that are critical to replace the completed thread. Also, as the children complete, more and more parents become critical and begin to migrate to the OOO_PROC.

2.3. Hybrid Schemes

The above schemes were also hybridized to create predictors that combine criticality metrics. To add CPAM or SPAM to any of the other predictors is uncomplicated, since they are disjoint from the regular scheduling interval mappings. However to hybridize the remaining mappings a hybrid score is created. For each mapping scheme, the results are collected as normal. During scheduling, each statistic undergoes a global normalization to re-scale it between 0 and 1 based on its global maximum. Once scaled, each thread's counter values are summed to create the hybrid score. The two threads with the highest hybrid scores are considered critical and are given to the heavyweight cores.

2.4. Existing Prediction Schemes

In addition to the new mapping schemes, we also evaluated several existing schemes. We target recent work on dynamic prediction with similar goals to our work to create a cache miss-rate scheme (L2) using recent cache miss rates, and an instruction count (IC) scheme using the recent instruction counts. The L2 scheme is motivated by [14], which found that the L2 miss rate correlates highly to the critical thread for several workloads. This scheme uses a similar base scheduling policy as our new schemes, except that the critical threads are the two threads with the highest L2 miss rate during the last scheduling interval.

IC uses the recent instruction counts to determine overall thread progress to predict criticality. Every scheduling interval, the instruction counts are reset and re-evaluated. This implementation is similar to [15], except barrier points are not used as the evaluation points. Many of the benchmarks evaluated have little to no barriers at which to remap. This scheme uses a similar base scheduling policy as our new schemes, except that critical threads are chosen as the two threads with the least progress in the last scheduling interval.

The baseline asymmetry aware mapping performs no dynamic migration. Instead it schedules a spawning or resuming thread on an OOO_PROC if one is available.

3. Implementation Details

To collect the data for the new predictors, hardware counters are added to each core for tracking. When control returns to the operating system during the regular scheduling interval, the scheduler collects

all of the counter values and calculates the necessary averages to make a global decision. The operating system may then migrate or move threads based upon the scheduling policies discussed in Section 2.

3.1. MAX_LEN hardware requirements

The MAX_LEN predictor must maintain the dependence graph in an iterative process and maintain the length of all paths in the dependence graph. Each instruction represents a node in the dependence graph, and each instruction has a single destination register. To maintain the graph, only the most recent writers of each register must be recorded. Therefore MAX_LEN maintains a structure with the number of entries equal to the number of registers (this may be stored alongside the registers in the register file, for example). Each entry stores the *cumulative path length* into the node as well as the number of paths leading into the node. Each entry requires 24 bits of storage for the total path length, and 8 bits of storage for the total path count. The table is reset and the average is stored in a collection register when an overflow occurs. For our configuration, this results in a table size of 512B per node (128 entries * 4 bytes per entry). This structure is updated at the decode stage when the registers are decoded.

3.2. MAX_DEP hardware requirements

The MAX_DEP predictor maintains a weighted dependence graph that considers output and anti-dependencies in addition to true dependences. The algorithm requires the latest use time of each register and last completion time to be maintained. A new instruction's depth is the maximum of the length resulting from anti-, output-, and true- dependences. Latency weights are based upon the opcode type. The latency values match the function unit's execution time for each opcode. For memory instructions, the latency is assumed to be 3 cycles for L1 hits, and 100 cycles on an L1 miss. This feedback is provided in the MEM stage when inspecting the L1. The table requires an entry for each register and each entry maintains the latest use times and completion times. This requires a table of size of 1kB per node (128 entries * 8 bytes per entry).

3.3. CPAM/SPAM hardware requirements

The CPAM mapping scheme only requires each thread to maintain the thread id of its parent. This is already standard information maintained in the operating system. Therefore, no additional hardware is required for the CPAM mapping scheme.

The SPAM mapping scheme can leverage either MAX_LEN or MAX_DEP to make its scheduling decisions on thread resume. This requires the appropriate table to be implemented, and the updates to the operating system of the most recent values every scheduling interval. Otherwise, no additional hardware is required.

4. Simulation Methodology

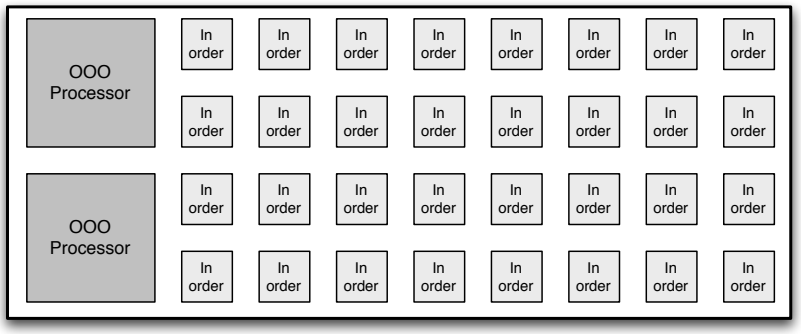

Figure 6 - Simulated Architecture Layout

To evaluate the various thread mapping schemes, an in-house cycle accurate simulator is used which is part of a larger, multi-agency-funded simulation framework being developed by the authors and other collaborators. The execution model leverages a MIPS based emulator that feeds the timing model. The system is an *execute-at-fetch* model except at synchronization points (LL, SC, and SYNC) where an *execute-at-execute* model is used. For these instructions, the emulator "peeks" without committing state, and then allows the timing model to resolve the lock winner. Thus thread interleaving is accurate to the simulated system rather than the host system.

The simulated architecture, shown in Figure 6, is a 34-core system with 32 in-order cores and 2 out-of-order cores. Out-of-order cores (OOO_PROCs) are assumed to take approximately 4x the die area of an in-order core (IO_PROC). Table 4 lists the parameters of the processor models, network, and cache hierarchy. Scheduling decisions are made when threads spawn, suspend, or resume. Additionally, every 100ms (100,000 cycles) thread re-scheduling occurs based upon the mapping scheme used. Similar to [15], the thread remapping penalty is assumed to be dominated by cache transfer, and thus that is the only penalty incurred for transfer. Cache warming is performed by setting the off-chip access latency to one cycle during warm-up to remove startup effects.

Table 4 - Simulation Parameters

In-order cores	32	Network Topology	Mesh (X-Y routing)
Out-of-order cores	2	Coherence Protocol	Blocking MESI
Memory Controllers	1		Inclusive Caches
Off-chip latency	200 cycles		Distributed shared directory
Out-of-order core			
L1 Cache	32 kB	L1 Cache Latency	3 cycles
L2 Cache	128 kB	L2 Cache Latency	8 cycles
Branch Predictor	Gshare		
ROB size	256 insns	Scheduling Interval	100 ms (100,000 cycles)
Issue width	4		
		Chip Frequency	1 Ghz
In-order core			
L1 Cache	32 kB	Virtual Channels	3
L2 Cache	64 kB	Buffer Entries per VC	2
Branch Predictor	Gshare	Router link latency	3 cycles
Issue Width	2		

Results are collected for a large subset of the PARSEC [22] benchmarks. Additionally, the SSCA2 OpenMP benchmark [25] is included to provide an additional graph traversal multi-threaded workload. The benchmark descriptions are summarized in Table 5. Both benchmark sets were run to completion to obtain accurate execution time measurements. For PARSEC, only the instructions in the region of interest were simulated and measured.

The benchmarks evaluated represent a variety of parallel programming patterns and thread behavior. Table 5 summarizes the primary pattern(s) used to program each benchmark. As is evident from the table, PARSEC has a diverse set of parallel patterns and thread behaviors. Therefore it is likely to provide promising potential and challenges for optimal thread mapping

Table 5 - Evaluated benchmark descriptions and patterns

PARSEC	
blackscholes	Blackscholes option pricing. Exhibits task parallelism.
bodytrack	Computer vision algorithm. Image is geometrically decomposed
canneal	Simulated annealing using a fine-grained pipeline pattern
dedup	Data compression using deduplication. Uses the pipeline programming model
ferret	Content similarity search using a pipeline model
fluidanimate	Animation using fluid dynamics. Combination of Divide and Conquer and Pipeline.
streamcluster	Online clustering of streaming data. Task parallel algorithm
swaptions	Monte Carlo simulation for options pricing. Exhibits task parallelism.
x264	H.264 video encoding using pipeline parallel programming model
SSCA	
SSCA #2	Graph analysis of a weighted directed graph. Uses a recursive data pattern.

5. Results and Analysis

In this section we present our findings regarding the performance of the proposed schedulers for the microbenchmarks, PARSEC, and SSCA#2. The performance is presented as the percent reduction in execution time (speedup) versus all in-order cores for all schemes.

5.1. Microbenchmark results

As discussed earlier, seven microbenchmarks were created to represent the five common parallel programming patterns. The execution time results for these runs are shown in Figure 7. The motivation section showed that the greatest potential for speedup exists in the divide and conquer (DandC, DandC2), pipeline, and recursive microbenchmarks. As the results indicate, these benchmarks exhibit the most intriguing behavior of the parallel patterns.

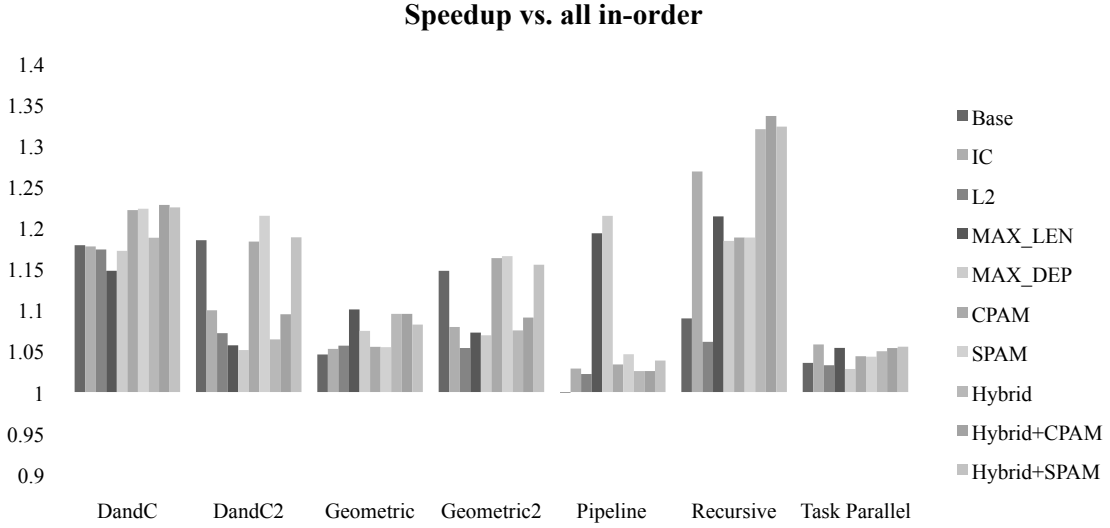

Figure 7 - Speedup results for pattern-centric microbenchmarks

Both divide and conquer benchmarks get the greatest benefit from the CPAM and SPAM mapping schemes. It is no surprise that CPAM performs well considering that it is targeting the thread spawn/exit inherent to the divide and conquer algorithm. SPAM is also effective since it will often predict parent threads are critical when they resume. SPAM works better on DandC2 because the length of the parent threads is greater than the children. The children are sorting a small subset of the array, while the parents are merging increasingly large sub-arrays. Complexity based thread remappings such as MAX_LEN and MAX_DEP are not as effective on divide and conquer patterns. This is largely because within thread

groups, complexity is fairly homogenous. Even when the divide and conquer is asymmetric in nature, the homogenous behavior stems from a difference in thread lengths, rather than thread complexity. Thus, IC is able to outperform complexity mappings for the divide and conquer microbenchmarks.

The geometric benchmark benefits from the MAX_LEN algorithm, which measures dependence length. However, this method is not nearly as effective for the smaller geometric2 benchmark. The difference arises from the heterogeneity in the geometric benchmark that results from one thread collecting data and doing extra work. Additionally, threads persist for a longer period because of the larger data set. This allows the benefit of a small amount of imbalance to grow slowly. Geometric2 does not respond well to any of the interval scheduling schemes. This is more typical of completely symmetric geometrically decomposed workloads. Since complexity and instruction counts will be very similar, constant remapping will cause extra cache thrashing that results in a slowdown. Furthermore, the shorter thread lengths from the small data set cause less chance to overcome the thread migration penalty.

The pipeline microbenchmark benefits only slightly from the SPAM mapping scheme. The reason this method is not as effective as MAX_LEN or MAX_DEP is due to the pipeline stages' lengths. When threads are shorter they are more sensitive to migration overheads. This benchmark, however, is complexity bound as two pipeline stages were deliberately written to have extra computational work. Because of this, MAX_LEN and MAX_DEP are therefore more effective at predicting criticality than IC. The hybrid scheme suffers from the inclusion of the IC metric, whose measurements conflict with the complexity predictors, resulting in more homogeneous criticality scores.

For all of the microbenchmarks, the reasons for fetch stalls were collected during simulations (Branch mispredictions, Memory dependence stalls, ALU dependence stalls, etc.). These results are omitted for space reasons; however, what they reveal is that the pipeline benchmark bottleneck is due to thread imbalance resulting in idle cores. By balancing the thread complexities, the pipeline benchmark is able to reduce the number of cycles cores spend idle, thus reducing the overall execution time.

The recursive data pattern consists of many varying length threads of similar complexity. Thus, the best mapping scheme would likely be the IC scheme, which is corroborated by the speedup results. The hybrid scheme outperforms IC, because in this benchmark, the complexity predictors prevent over-

migration and over-prediction by the IC scheme. The stall distribution for the IC scheme reveals additional memory access stalls. A deeper analysis revealed that the increase is due to an increased L1 miss rate (1.2% for IC and 0.05% for Hybrid). Thus, migration is causing thrashing in the L1 that is slowing down overall performance.

The task parallel benchmark has largely similar results across all schemes, due to its homogeneous thread behavior, which has low potential for thread remapping benefit. A more appropriate mapping scheme for this type of parallel pattern would focus on fairness or overall power savings rather than aggressive remapping.

5.2. Full Benchmark Results

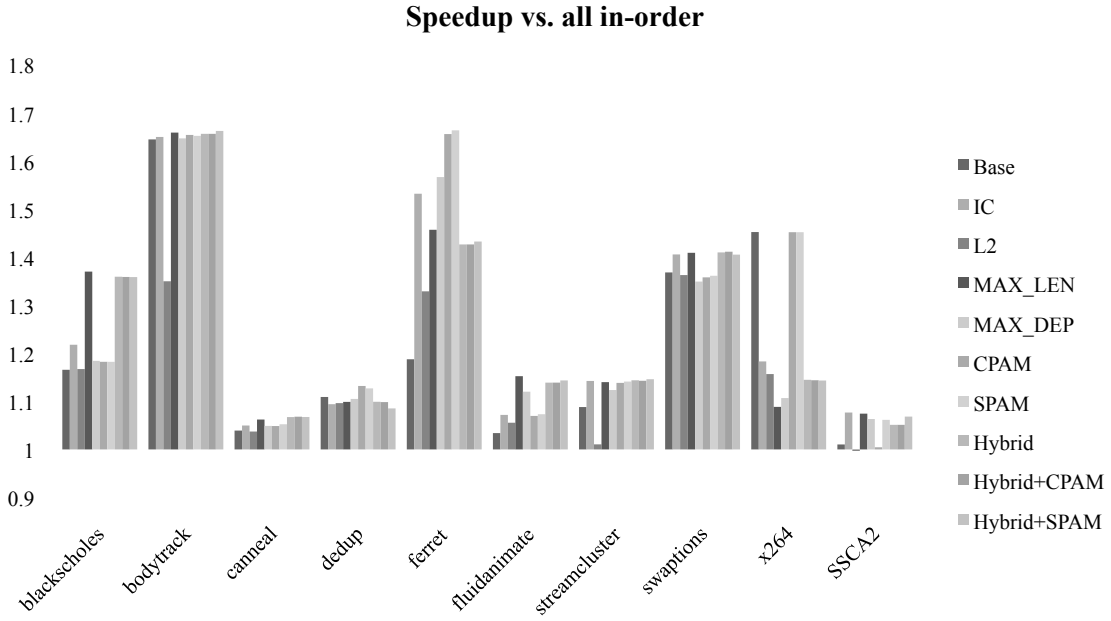

Figure 8 - Speedup results for PARSEC and SSCA#2

The results of the full benchmark runs are shown in Figure 8. The varying performance illustrates that thread remapping is highly dependent upon the programming pattern and data behavior of the benchmarks. The SSCA benchmark works best for the IC and MAX_LEN mapping patterns. SSCA is an example of a recursively parallelized benchmark, and the results are in line with the expected behavior shown in the recursive microbenchmark. The disparity between thread length and complexity is lower than in the associated microbenchmark, thus the performance of IC and MAX_LEN are more similar.

Additionally, there are a large percentage of stalls due to the pipeline flushing from LL, SC, and SYNC instructions. These stalls limit the overall performance gains of the SSCA benchmark and explain why even the best thread balancing schemes for the recursive data pattern can only achieve slightly less than 10% improvement over the baseline.

Blackscholes is a task parallel style benchmark, which already has mostly homogeneous thread instruction counts. Therefore, mapping schemes such as IC should not perform well, as the predicted critical threads will change constantly due to small variations. The MAX_LEN mapping performs surprisingly well, however. An investigation of the distribution of stall cycles found that in the MAX_LEN experiments, stalls due to memory latency were reduced, specifically time spent on L1 misses. This result suggests that complexity imbalance was more stable than instruction counts. In the blackscholes algorithm, there are two types of options, put options and call options. In the input data set, each option represents about half of the input data. Code inspection reveals that call options require some extra data manipulation compared to put options. The MAX_LEN predictor is able to detect this more readily than IC, which fails due to the balanced instruction counts.

The bodytrack benchmark is a geometrically decomposed workload that is balanced, and thus little performance gain is found through thread remapping. The L2 scheme, however, performs poorly as it does for most of the PARSEC workloads. We found that in more traditionally balanced geometrically decomposed workloads, such as SPLASH-2, the L2 scheme performs much better. However the homogeneous task behavior of SPLASH-2 presented little opportunity for any of the dynamic schemes.

The traditional pipeline parallel benchmarks in PARSEC: dedup, ferret, and x264, all perform the best using the SPAM mapping scheme. Unlike the pipeline microbenchmark, the cost of mid-thread migration causes too much cache thrashing. A look at the stall type distribution revealed that by preventing mid-thread migration the number of stalls due to memory was reduced by as much as 33%, which translates to a reduction in the number of overall stalls by 25%. In x264, this behavior was the most pronounced, as the parallel section accounts for a small portion of the benchmark (data input and output), and the threads are short and thus very sensitive to migration.

SPAM works exceptionally well for ferret due to the nature in which threads are managed. Ferret makes use of thread pools for each pipeline stage. The initial stage, which reads in data, is given only a single thread, while the other stages are each given up to 32 threads to work with. The passing of data between the threads results in more suspend/resume events for CPAM to decide upon. Secondly, by having many threads per pipeline stage, threads from the same stage will stay at the top of the complexity list. The SPAM mapping scheme limits the amount of thrashing between these threads by only scheduling threads from within a stage to an OOO_PROC when they begin, whereas MAX_DEP will cause threads for the same stage to thrash from having similar complexities.

Fluidanimate is an interesting case as it exhibits thread heterogeneity and benefits from complexity mapping. Typically, fluid dynamics problems are geometrically decomposed as particles are distributed among threads, and each thread works on a fixed subset of particles. However, fluidanimate divides the frame to animate into segments and particles move through fixed segments from frame to frame. The particles are therefore not evenly divided among the threads. This creates a thread imbalance that is similar to a pipeline style pattern. Fluidanimate is a different style of pipeline pattern than either the microbenchmark or any of the pipeline benchmarks discussed above. Instead of a pure feed-forward pipeline, pipeline stages are interconnected in a grid and data flows around the grid. The varying number of particles each thread must process causes a complexity imbalance that is exploited by the MAX_LEN predictor. Therefore, like the pipeline microbenchmark, which was also complexity bound, MAX_LEN best predicts the critical thread. Overall, our schemes achieve upwards of 40% performance improvement over the base asymmetric mapping and a 25% improvement over current dynamic schemes.

6. Related Work

Load balancing and thread scheduling for asymmetric processors is a heavily studied field. Original work focused on the benefit of heterogeneity as a low power alternative for CMPs that is adaptive to multiple program types [7, 8]. It was found that only a few powerful cores are needed to achieve near-optimal performance gains for a wide class of programs. Leveraging this potential, however, requires a mapping scheme that best utilizes the powerful cores. Several works have analyzed static schemes based on profiling or a mathematical model [8-10, 12]. Additionally [26] investigated the trade-off between

receiver-initiated and sender-initiated load balancing policies. In receiver-initiated polices, the lightly loaded core seeks out a task to steal. In sender-initiated policies, the heavy loaded core seeks a core to offload take some of its tasks. The MAX_DEP and MAX_LEN policies in this paper are a type of sender-initiated policy, while CPAM and SPAM are receiver-initiated policies.

Thread mapping for heterogeneity provides power benefits as well as performance. Many prior works have focused on leveraging thread mapping to achieve optimal performance for a fixed power budget [27-29]. Also, asymmetry has been leveraged in the multi-programmed domain for optimal program mapping [16, 29]. In addition to fixed asymmetry, additional studies have investigated dynamic architecture reconfiguration through voltage-frequency scaling and reconfigurable pipelines [30].

Dynamic thread mapping schemes have proposed several mechanisms for predicting, migrating, and scheduling the critical thread. In [11] batches of tasks are scheduled as a group based on the thread throughput. Because throughput is tied to the core type running the thread, this creates the potential for thrashing as threads may be overly remapped. In [19], task graphs are leveraged to group subtasks and schedule the group on the appropriate core type. [13] extends traditional operating system scheduling to weight the load factor of the scheduling queues by the processing power of the core. [14] and [16] observe that thread criticality can be predicted using the L1 and L2 miss rates. Thread criticality predictors in [14] leverage hardware counters similar to the ones evaluated in this work based upon cache miss rates. Also, especially in homogenous workloads, instruction count can provide a good indicator of thread criticality. In [15] the length between synchronization regions is recorded and the instruction count between previous intervals is used to measure criticality. The IC based scheme used for comparison in this work represents this scheme.

Pattern aware thread mapping has also been studied in a couple of prior works. [31] provides an API for programmers using a pipeline model to delimit pipeline stages. Additionally this work finds that the communication and computation behavior of a pipeline is relatively stable. This fact was the basis for the SPAM predictor used in this work. Recently [18] has leveraged a pipeline friendly API to profile which pipeline stage is the LIMITER stage. This work attempts to allocate more cores to the LIMITER stage to improve load balancing and maintain a higher throughput. This approach is solving the same problem as

our SPAM mapping algorithm; however, our SPAM mapping algorithm is agnostic of the pipeline boundaries and requires no software modification.

7. Conclusion and Future Work

In this work we have proposed four novel, hybrid hardware/software, pattern-based thread mapping predictors. Two predictors, MAX_DEP and MAX_LEN focus on thread remapping during the regular scheduling interval. They perform well on criticality caused thread heterogeneity. Our other two predictors, CPAM and SPAM make effective thread mapping decisions during spawn/exit and suspend/resume, respectively. Overall these predictors were shown to improve the base asymmetry aware scheme by as much as 40% and existing dynamic schemes by as much as 23%. Furthermore, this paper provided a better understanding of the relationship between thread behavior and the parallel pattern used to develop the workload. We found that pipeline parallel, divide and conquer, and recursive data exhibit the most heterogeneity and thus benefit the most from thread scheduling.

Future work will focus on extending the current predictors to reduce thrashing among a group of similar threads and expanding the architecture design space. The 100 ms reset interval was chosen so that idle threads during the last scheduling interval are not considered. However, pipeline benchmarks typically exhibit stable behavior [31], and a periodic refresh of the dependence counters may cause false criticality predictions. Future work will study adding aging mechanisms to address patterns that exhibit relatively stable behavior throughout their execution.

8. Bibliography

[1] *Intel Core 2 Quad Core Processors*. Available: http://www.intel.com/products/processor/core2quad/

[2] *Intel Xeon Processor*. Available: http://www.intel.com/p/en_US/products/server/processor/xeon6000

[3] *Nvidia Tesla*. Available: http://www.nvidia.com/object/product_tesla_c1060_us.html

[4] K. Asanovic, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California, Berkeley, Berkeley, CA, Technical ReportDec 2006.

[5] P. Kongetira*, et al.*, "Niagara: a 32-way multithreaded Sparc processor," *Micro, IEEE,* vol. 25, pp. 21-29, 2005.

[6] D. C. Pham*, et al.*, "Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor," *Solid-State Circuits, IEEE Journal of,* vol. 41, pp. 179-196, 2006.

[7] R. Kumar*, et al.*, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, 2004, pp. 64-75.

[8] R. J. O. Figueiredo and J. A. B. Fortes, "Impact of heterogeneity on DSM performance," in *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, 2000, pp. 26-35.

[9] N. G. Shivaratri and P. Krueger, "Two adaptive location policies for global scheduling algorithms," in *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, 1990, pp. 502-509.

[10] D. A. Menasce*, et al.*, "Processor assignment in heterogeneous parallel architectures," in *Parallel Processing Symposium, 1992. Proceedings., Sixth International*, 1992, pp. 186-191.

[11] L. Chin and L. Sau-Ming, "An adaptive load balancing algorithm for heterogeneous distributed systems with multiple task classes," in *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, 1996, pp. 629-636.

[12] H. Oh and S. Ha, "A Static Scheduling Heuristic for Heterogeneous Processors," presented at the Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II, 1996.

[13] T. Li*, et al.*, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," presented at the Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Reno, Nevada, 2007.

[14] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," presented at the Proceedings of the 36th annual international symposium on Computer architecture, Austin, TX, USA, 2009.

[15] N. B. Lakshminarayana*, et al.*, "Age based scheduling for asymmetric multiprocessors," presented at the Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon, 2009.

[16] J. C. Saez*, et al.*, "A comprehensive scheduler for asymmetric multicore systems," presented at the Proceedings of the 5th European conference on Computer systems, Paris, France, 2010.

[17] J. L. Ortega-Arjona, *Patterns for Parallel Software Design*. West Sussex: Wiley, 2010.

[18] M. A. Suleman*, et al.*, "Feedback-directed pipeline parallelism," presented at the Proceedings of the 19th international conference on Parallel architectures and compilation techniques, Vienna, Austria, 2010.

[19] L. De Giusti*, et al.*, "AMTHA: An Algorithm for Automatically Mapping Tasks to Processors in Heterogeneous Multiprocessor Architectures," in *Computer Science and Information Engineering, 2009 WRI World Congress on*, 2009, pp. 481-485.

[20] R. C. Agarwal*, et al.*, "A high performance parallel algorithm for 1-D FFT," in *Supercomputing '94. Proceedings*, 1994, pp. 34-40.

[21] J. P. Singh*, et al.*, "Scaling parallel programs for multiprocessors: methodology and examples," *Computer,* vol. 26, pp. 42-50, 1993.

[22] C. Bienia*, et al.*, "The PARSEC benchmark suite: characterization and architectural implications," presented at the Proceedings of the 17th international conference on Parallel architectures and compilation techniques, Toronto, Ontario, Canada, 2008.

[23] T. G. S. Mattson, Beverly A.; Massingill, Berna L., *Patterns for Parallel Programming*: Addison-Wesley, 2004.

[24] J. H. Patel and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays," presented at the 25 years of the international symposia on Computer architecture (selected papers), Barcelona, Spain, 1998.

[25] D. Bader and K. Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," in *High Performance Computing – HiPC 2005*. vol. 3769, D. Bader*, et al.*, Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 465-476.

[26] D. L. Eager*, et al.*, "A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract)," *SIGMETRICS Perform. Eval. Rev.,* vol. 13, pp. 1-3, 1985.

[27] M. Annavaram*, et al.*, "Mitigating Amdahl's law through EPI throttling," in *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, 2005, pp. 298-309.

[28] A. Fedorova*, et al.*, "Maximizing power efficiency with asymmetric multicore systems," *Commun. ACM,* vol. 52, pp. 48-57, 2009.

[29] R. Kumar*, et al.*, "Core architecture optimization for heterogeneous chip multiprocessors," presented at the Proceedings of the 15th international conference on Parallel architectures and compilation techniques, Seattle, Washington, USA, 2006.

[30] M. Pericas*, et al.*, "A Flexible Heterogeneous Multi-Core Architecture," presented at the Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, 2007.

[31] W. Thies*, et al.*, "A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs," presented at the Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007.