Feedback Driven Pipelining

M. Aater Suleman   Moinuddin Qureshi   Khubaib   Yale N. Patt

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2010-001
March 2010

This page is intentionally left blank.

Feedback-Directed Pipelining for Multi-threaded Workloads

Abstract

Extracting performance from Chip Multiprocessors requires that the application be parallelized. A common software technique to parallelize loops is pipeline parallelism *in which the programmer/compiler splits each loop iteration into stages and each stage runs on a certain number of cores. It is important to choose the number of cores for each stage carefully because the core-to-stage allocation determines performance and power consumption. Finding the best core-to-stage allocation for an application is challenging because the number of possible allocations is typically quite large, and the best allocation changes with the input set and machine configuration.*

This paper proposes Feedback-Directed Pipelining (FDP)*, a software framework that chooses the core-to-stage allocation at run-time. FDP first maximizes the performance of the workload and then saves power by reducing the number of active cores, without impacting performance. Our evaluation of FDP on a real 8-core SMP system (2x Core2Quad) shows that FDP provides an average speedup of 4.2x which is significantly higher than the 2.3x speedup obtained with a practical profile-based allocation. We also show that FDP is robust to changes in machine configuration and input set variation.*

1. Introduction

Modern processors tile multiple cores on a single chip to improve concurrency. As processor frequency has slowed down, and the per-core performance is improving at a much slower pace than before, applications will focus on exploiting parallelism for performance growth. Improving performance of a single application using such a multiprocessor system requires that the application be divided into threads. Threads concurrently execute different portions of the same problem, thereby improving performance. As applications tend to spend most of their time in executing loops (or recursive kernels, which can often be converted into loops), we focus primarily on extracting parallelism within loops.

Pipeline parallelism is a popular software approach to split the work in a loop among threads. In pipeline parallelism, the programmer/compiler splits each iteration of a loop into multiple work-quanta where each work-quantum executes in a different pipeline stage. Recent research has shown that pipeline parallelism is applicable to many different types of workloads, e.g, streaming [7], recognition-mining-synthesis workloads [2], compression/decompression [10], etc. In pipeline parallel workloads, each stage is allocated one or more *worker* threads and an *in-queue* which stores the work quanta to be processed by the stage.[1] A worker thread pops a work quanta from the in-queue of the stage it is allocated to, processes the work, and pushes the work on the in-queue of the next stage.

Figure 1(a) shows a loop which has N iterations. Each iteration is split into 3 stages: A, B, and C. Figure 1(b) shows a flow chart of the loop. The three stages of the ith iteration are labeled Ai, Bi, and Ci. Figure 1(c) shows

[1] We always execute one thread per core and therefore use cores and threads interchangeably.

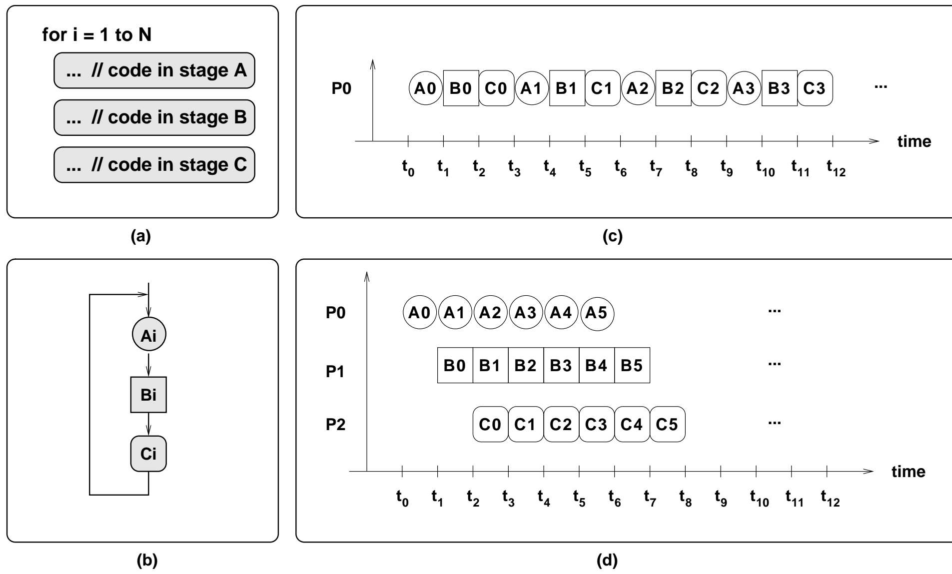

Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.

how this loop gets executed sequentially on a single processor. The time t_0 is the start of iteration 0 of the loop. The time t_3 is the end of iteration 0, and the start of iteration 1, and so on. Figure 1(d) shows how this program gets executed using pipeline parallelism on three processors. Each core works on a separate part of the iteration (P0 executes stage A, P1 executes stage B, and P2 executes stage C), and the iteration gets completed as it traverses from left to right, and top to bottom. Note that we show for simplicity that each stage has one core but it is possible to allocate multiple cores per stage or share a core among stages. When multiple cores are assigned to a stage, they all feed from the in-queue assigned to the stage and execute different work-quanta concurrently. In fact, a key design decision in developing code using pipeline parallelism is to determine the total number of stages and the number of threads (cores) which are allocated to each stage.

Pipeline parallelism can be implemented using a *Symmetric Pipeline*, in which each stage has an equal number of cores. The key problem with a symmetric pipeline is that each stage can perform disparate parts of the work, each with widely varying latency. If each stage takes different time to execute its work quanta, then all stages other than the slowest one remain under-utilized. This problem can be solved by using an *Asymmetric Pipeline* that allocates a different number of cores to each stage. However, determining the core allocation per stage for optimal performance is a non-trivial task, as the latency per stage is a function of input set and machine configuration and may change between different phases of a given program. Furthermore, given that not all stages benefit equally from each extra core, the core allocation must be based not only on latency but also on how well a stage utilizes execution resources. Allocating more cores to a stage than required to saturate its performance wastes power and sometimes reduces performance.

The core-to-stage allocation can be done statically using profile information. However, profiling information

is typically dependent on input set and is applicable only for a particular machine. When the input set or the machine changes, the decisions based on profile information may not be meaningful. Furthermore, searching through all the combinations of core-to-stage allocations may be impractical given that the number of possible allocations increases combinatorially with the number of cores.

To overcome these limitations of pipeline parallelism, this paper proposes *Feedback-Directed Pipelining (FDP)*, a framework that can execute pipeline parallel workloads in a high performance and power-efficient manner. For dynamic core-to-stage allocation, FDP leverages the key insight that the performance of a pipeline is limited by the execution rate of the slowest stage. Thus, highest performance can be achieved only when maximum possible resources are allocated for the acceleration of the slowest stage. FDP samples the execution to measure latencies of each stage and uses a hill-climbing algorithm to determine core-to-stage allocation.

Once the slowest stage has been accelerated to the maximum, FDP can slow down the other stages to save resources. For example, allocating the same core to two different stages which are utilizing their cores less then 50%. Combining stages frees up cores which are either used to improve performance of other stages or yielded to the Operating system. The operating system can either assign these cores to other programs or turn them off to save power.

Previous researchers have also proposed mechanisms to choose the number of threads per stage statically [15, 16, 13, 6] or dynamically [9]. The static mechanisms have the shortcoming that they cannot take the input set, machine configuration, or scalability of stages into account. The previously proposed dynamic mechanisms make simplistic assumptions about scalability of stages and are limited to workloads where stages are relatively balanced and have similar characteristics. Unlike these previously proposed techniques, FDP is a general mechanism which makes no assumption about the stages' execution time or their scalability. FDP is a dynamic mechanism which measures the run-time and infers the scalability of each stage via hill-climbing. Thus, FDP can adapt to changes in input set and machine configuration and is applicable to all pipeline workloads, even where stages are heavily imbalanced.

We evaluate FDP on a real 8-core Core2Quad SMP using 9 workloads (experimental methodology is shown in Section 4). FDP provides an average speedup of 4.2x which is significantly higher than the 2.3x speedup obtained with a practical profile-based allocation. FDP also reduces the average number of active cores by 12.5%. We also evaluate FDP on a 16-core Barcelona system and show that FDP continues to provide significant performance, while reducing the number of active cores. Furthermore, we show that FDP is also applicable to workloads parallelized using *Work Sharing*, an alternative programming paradigm.

FDP is a software technique and does not require any hardware changes. It measures the execution time using existing instructions to read the processor time stamp counter. We develop a software library which contains routines that implement the FDP algorithm. These routines measure the execution time, determine the

core-to-stage allocation, and enforce these allocations. The library abstracts the details and provides a simple interface to reduce programmer intervention.

2. Motivation

As chip-multiprocessors become common, programmers will resort to multi-threading as means to improve performance. Improving performance of loops using multi-threading requires distributing the work among threads. An effective approach to distributing work is pipeline parallelism. Pipeline parallelism has been shown to increase parallelism, improve cache locality, and increase power efficiency [7].

2.1. Pipeline Programming Model

Pipeline parallel workloads extract parallelism at two different levels: within the same iteration of a loop and between different iterations of a loop. To execute the loop as a pipeline, the programmer/compiler divides an iteration of a loop into distinct stages of work. All stages are scheduled such that they can run concurrently. An iteration enters the pipeline and "flows" through the pipeline stages as different stages operate on it. The iteration is complete once it leaves the last stage in the pipeline.

In a pipeline program, each stage is assigned a work-queue, which we call its *in-queue* and one or more *worker* threads. Each entry in the in-queue is the pointer to an iteration. An entry pointing to iteration i in the work-queue for stage s signifies that stage s of iteration i is to be executed. Each worker thread can also be assigned to multiple stages and it can process execution requests from the in-queues of any of the stages it is assigned to. The worker thread dequeues an iteration from the in-queue of one of the stages it is assigned to, processes the stage, and enqueues the iteration in the in-queue of the next stage. For example, let a worker thread w be assigned to stage s. Now suppose that when w dequeues a request from the in-queue of stage s, it finds iteration i. w will then run stage s of iteration i and then add i to the in-queue of stage $s + 1$.

```
1:  while (!DONE)
          // GetNextStage(): Pick a stage to execute
          // A stage is chosen in round-robin fashion from
          // the set of stages which satisfy two criteria:
          //    -Stage must be assigned to the worker thread
          //    -Stage must have a non-empty in-queue
          // The thread waits if all such stages have empty in-queues
2:          stage = GetNextStage()
3:          Pop an iteration i from stage's in-queue
4:          Run stage for iteration
5:          Push the iteration to the in-queue of its next stage
```

Figure 2. The worker loop.

Figure 2 shows the source code of a generic worker thread often used in a pipeline. The worker thread runs in a loop until the program is complete, i.e., all iterations have been processed. In each iteration of the worker thread loop, the thread picks the stage to run: stages are chosen in round-robin fashion from the set of stages who are assigned to the worker thread and whose in-queue is non-empty. If all the stages mapped to a worker

thread have an empty in-queue, the worker thread polls on these in-queues until one of them is non-empty. Once the worker thread has found a stage with a non-empty in-queue, it dequeues an iteration from the queue, executes the stage for the iteration, and then enqueues the iteration in the in-queue of the iteration's next stage. We now explain pipeline parallelism with an example application.

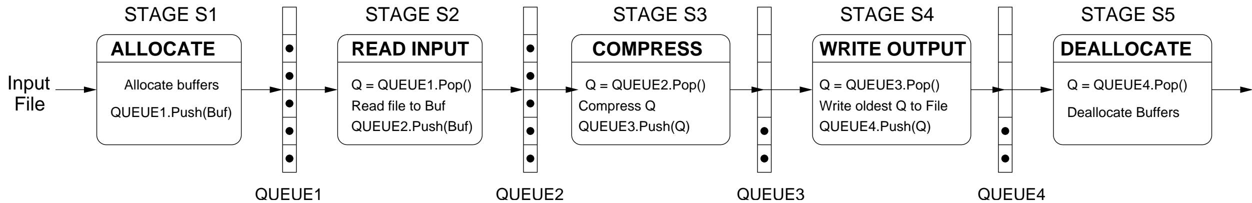

Figure 3. File compression algorithm executed using pipeline parallelism

Consider a kernel from the workload `compress`. This kernel compresses the data in an input file and writes it to an output file. Each iteration of this kernel reads a block from the input file, compresses the block, and writes the compressed block to the output file. Figure 3 shows the pipeline of this kernel. Stage S1 allocates the space to save the uncompressed and the compressed block. S2 reads the input and S3 compresses the block. When multiple threads/cores are allocated to each stage, iterations in a pipeline can get out of order. Since blocks must be written to the file in-order, S4 re-orders the quanta and writes them to the output file. S5 deallocates the buffers allocated by S1. This kernel can execute on a 5-core CMP such that each stage executes on one core. At any point in time, cores will be busy executing different portions of five different iterations, thereby increasing performance. In reality, when the pipeline executes, cores executing different stages of a pipeline often wait on other cores and remain idle. This limits concurrency and reduces performance. There are two common sources of this inefficiency.

2.2. Variation in Throughput

We define *throughput* of a pipeline stage as the number of iterations processed in a given amount of time. Thus, the throughput τ_i of a pipeline stage i can be defined as:

$$\tau_i = \frac{Num\ Iterations\ Processed}{Time} \tag{1}$$

The overall throughput, τ, of the whole pipeline is limited by the throughput of the slowest stage of the pipeline. Therefore:

$$\tau = MIN(\tau_0, \tau_1, \tau_2, ...) = \tau_{min} \tag{2}$$

Thus, for example, if the slowest stage of the pipeline for compression shown in Figure 3 is S3 (compress), then performance will be solely determined by the throughput of S3. The variation in throughout among stages also dictates the power efficiency of the pipeline. Let $LIMITER$ be the stage with the lowest throughput. Then stages other than the LIMITER will wait on the LIMITER stage and their cores will be under-utilized.

Therefore, the more the variation in the execution latencies of the pipeline stages, the more is the under utilization of cores, which leads to wasted on-chip power.

2.3. Limited Scalability

A common method used to increase the throughput of the LIMITER stage is to increase the number of cores allocated to it. However, more cores help if and only if the LIMITER stage scales with the number of cores (increasing the number of allocated cores increases its throughput). Unfortunately, throughput of a stage does not always increase with the number of cores due to contention for shared data (i.e. data-synchronization, cache-coherence) and contention for shared resources (e.g. caches and off-chip bandwidth). When a stage does not scale, allocating more cores to the stage either does not improve its throughput or can in some scenarios reduce its throughput [20]. Thus, once a stage becomes limited, the additional cores dissipate on-chip power without contributing to performance.

2.4. Need for Runtime Learning

The core-to-stage allocation can be done statically using profile information. However, profiling information is typically dependent on the input set and is applicable only for a particular machine. When the input set or the machine configuration changes, the decisions based on profile information may no longer be meaningful. Furthermore, searching through all the combinations of core-to-stage allocation may be impractical given that the number of possible allocations increase combinatorially with cores. For a system with C cores, a pipeline with S stages would have number of possible allocations given by($S \geq 2$ and $C \geq S$):

$$Num.\ Possible\ Allocations = \frac{\prod_{i=1}^{S-1}(C-i)}{\prod_{i=1}^{S-1} i} \tag{3}$$

For the above equation, we assume that each stage gets at least one core, all cores are allocated, and a core is not shared between multiple stages[2]. Table 1 shows the total number of combinations when the number of stages in a pipeline is varied from 2 to 8 for an 8-core, 16-core, and 32-core system.

Table 1. Num. allocations for an S stage pipeline.

Stages	2	3	4	5	6	7	8
8-Core	7	21	35	35	21	7	1
16-Core	15	105	455	1365	3003	5005	6435
32-Core	31	465	4495	31K	170K	736K	2.6M

Thus, the brute-force method of searching through the entire search space becomes impractical, especially as the number of cores continues to increase for future systems. An intelligent scheme that can learn the core-to-stage allocation at runtime can obtain close to (or better than) static profile-based allocation and will be robust

[2]Note that there are far more combinations possible if the above mentioned constraints are relaxed.

to input set and machine configuration. In the next section, we propose such a dynamic scheme.

3. Feedback-Directed Pipelining

The performance and power-efficiency of pipeline parallelism can be improved by making two key observations. First, the overall performance is dictated only by the LIMITER stage, hence more resources must be invested to improve the throughput of the LIMITER stage. Second, since the overall performance is not limited by the stages other than the LIMITER stage, withdrawing excess resources from these stages can improve power efficiency without impacting overall performance. We use these insights to propose *Feedback-Directed Pipelining (FDP)*, a parallelization framework that can achieve both high performance and low power.

3.1. Overview

FDP uses runtime information to choose core-to-stage allocation for best overall performance and power-efficiency. Figure 4 shows an overview of the FDP framework.

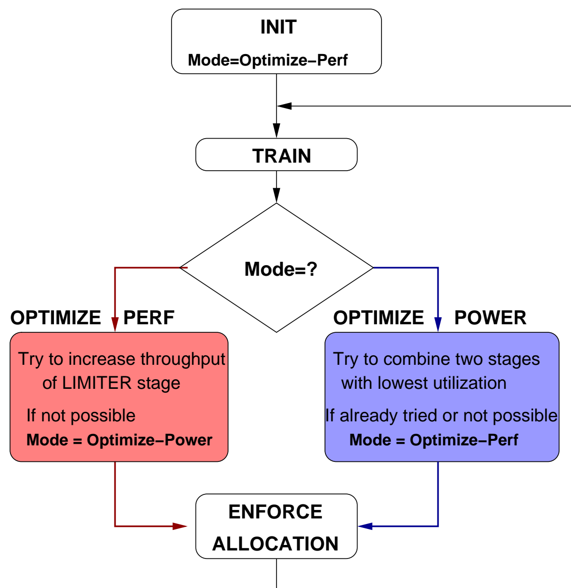

Figure 4. Overview of FDP.

FDP operates in two modes: one that optimizes performance (`Optimize-Perf`) and other that optimizes power (`Optimize-Power`). Initially, each stage in the pipeline is allocated one core. FDP first tries to achieve the highest performance, and then it tries to optimize power. FDP is an iterative technique that contains three phases: training, re-allocation of cores to stages, and enforcement of the new allocation. The training phase gathers runtime information for each stage of the pipeline, and is helpful in determining the throughput and core utilization of each stage. Based on this information, the performance-optimization mode identifies the LIMITER stage and tries to increase its throughput by allocating more cores to it. When it can no longer improve performance (as there may be no spare cores or adding cores does not help improve performance) FDP switches to power-optimization mode. In this mode, FDP tries to asssign the stages with lowest utilization to

Stages:	P0	P1	P2	Avg. Execution Time	Throughput
S0 :	(3K,3)			1K	1/1K
S1 :		(12K, 3)		4K	1/4K
S2 :	(9K, 1)		(21K,2)	10K	**1/5K**

Figure 5. Sample output from Train for a pipeline with three stages (S0, S1, S2) on a 3-core machine. Each entry is a 2-tuple: (the sum of time measurements, the number of time measurements) taken for each core-stage pair. Blank entries contain (0,0).

one core, as long as the combined stage does not become the LIMITER stage. The core thus saved can be used to improve performance or turned off to save power. Every time FDP chooses a new core-to-stage allocation, it enforces the new allocation on the pipeline at the end of the iteration. We now explain each part of FDP in detail (the psuedo-code of the FDP library is shown in the Appendix).

3.2. Train

The goal of the training phase is to gather runtime statistics about each stage. To measure execution time of each stage, the processor's cycle count register is read at the beginning and end of each stage. Instructions to read the cycle count register already exist in current ISAs, e.g., the `rdtsc` instruction in the x86 ISA. The difference between the two readings at the start and end of the stage is the execution time of the stage. This timing information is stored in a two-dimensional table similar to the one shown in Figure 5. The rows in the table represent stages (S0-S2) and columns represent cores (P0-P2). Each entry in this table is a 2-tuple: the sum and the number of time measurements taken for the corresponding core-stage pair. For each measurement taken, Train adds the measured time to the sum of measured times of the core-stage pair and increments the corresponding number of measurements. For example, if Train measures that executing S0 on P0 took 4K cycles, then it will modify the entry corresponding to S0 and P0 in Table 5 to (7K,4) i.e. (3K+4K, 3+1). Note that if a stage is not assigned to a core, the entry corresponding to the core-stage pair remains (0,0). For example, since S1 is only assigned to P1 and not to P0 and P2, its enteries for P0 and P2 are 0. We limit the overhead of measuring the timing information via sampling: we measure it once every 128th work-quanta processed by the stage.

3.3. Performance-Optimization

The goal of the performance-optimization mode is to change the core-to-stage allocation in order to improve overall performance. When the mode of operation is performance-optimization, one of the threads invokes this phase once every 2K iterations or 100K processor cycles, whichever is earlier[3]. The phase takes as its input the information collected during training, a table similar to Figure 5. The phase first computes the average execution time of all stages. The average execution time of a stage is the sum of all timing measurements recorded in the table for that stage divided by the total number of measurements for that stage. For example, for the table

[3]We choose these values empirically.

shown in Figure 5, the average execution time of stage S2 is 10K cycles computed as (9K+21K)/(1+2). The phase next computes the throughput of each stage as the number of cores assigned to the stage divided by the stage's average execution time (e.g., throughput of S2, which runs on two cores, is 2/10K, i.e., 1/5K). The stage with the lowest throughput is identified as the LIMITER (S2 is the LIMITER stage in our example). If there are free cores in the system, FDP allocates one of them to the LIMITER. The cores assigned to the LIMITER stage execute in parallel and feed from the in-queue assigned to the LIMITER stage.

To converge to the best decision, it is important that the core-to-stage allocations, that have already been tried, are not re-tried. FDP filters the allocations by maintaining the set of all allocations which have been tried. A new allocation is only enforced if it has not been tried before except when FDP is reverting back to a previous allocation that is known to perform similar to (or better than) the current allocation, while using fewer cores.

FDP increases the number of cores of the LIMITER stage with an implicit assumption that more cores lead to higher throughput. Unfortunately, this assumption is not always true; performance of a stage can saturate at a certain number of cores and further increasing cores wastes power without improving performance. To avoid allocating cores that do not improve performance, FDP always measures and stores the performance of the previous allocation. Every time FDP assigns a new core to the LIMITER stage, it compares the new performance with the performance of the previous allocation. If the new performance is higher than the performance with the previous allocation, FDP allocates another core to the LIMITER stage. However, if the new performance is lower than the performance with the previous allocation, FDP reverts to the previous allocation and switches to power-mode.

3.4. Power-Optimization

The goal of this mode is to reduce the number of active cores, while maintaining similar performance. When the mode of operation is power-optimization, this phase is invoked once every 2K iterations or 100K processor cycles whichever is earlier. This phase uses the information collected during training to compute the throughput of each stage. To improve power-efficiency, the stages with the highest throughput allocated to the two cores can be combined to execute on a single core, as long as the resulting throughput is not less than the throughput of the LIMITER stage. This optimization frees up one core which can be used by another stage for performance improvement or turned off for saving power. This process is repeated until no more cores can be set free. At this point, FDP reverts to performance mode.

3.5. Enforcement of Allocation

FDP changes the allocation of cores to stages dynamically. To facilitate dynamic allocation we add a data structure which stores for each core the list of stages allocated to it. The core processes the stages allocated to it in a round-robin fashion. FDP can modify the allocation in three ways. First, when a free core is allocated to the LIMITER stage, the LIMITER stage is added to the list of the free core. Second, when a stage is removed

from a core, it is deleted from the core's list. Third, when stages on two different cores are combined on to a single core, the list of one of the cores is merged with the list of other core and emptied.

3.6. Programming Interface for FDP

The FDP library itself handles the code for measuring and recording the execution time of each stage. It also maintains sampling counters for each allocation to limit instrumentation overhead. It automatically invokes performance-optimization or power-optimization phases at appropriate times without programmer intervention. To interface with this library, the programmer must insert in the code the four library calls shown in Figure 6.

```
void FDP_Init (num_stages)
void FDP_BeginStage (stage_id)
void FDP_EndStage (stage_id)
int    FDP_GetNextStage ()
```

Figure 6. FDP library interface.

The `FDP_Init` routine initializes storage for FDP and sets the mode to optimize performance. The training phase of FDP reads the processor's cycle count register at the start and end of every stage. To facilitate this, a call to `FDP_BeginStage` is inserted after the work-quanta is read from the respective queue and before it is processed. Also, a call to `FDP_EndStage` is inserted after the processing of the quanta is complete but before it is pushed to the next stage. The arguments of both function calls is the stage id. Once a core completes a work-quanta, it needs to know which stage it should process next. This is done by calling the `FDP_GetNextStage` function. FDP obtains the id of the core executing an FDP function by invoking a system call.

FDP only requires modifications to the code of the worker thread in a pipeline program, not the code which does the actual computation for the stage. Thus, FDP can be implemented in the infrastructures commonly used as foundation for implementing pipeline programs, e.g., Intel Threading Building Blocks [10].

```
1:  FDP_Init ()
2:      while (!DONE)
3:          stage_id = FDP_GetNextStage ()
4:          Pop an iteration i from the stage's in-queue
5:          FDP_BeginStage (stage_id)
6:          Run the the stage of that iteration
7:          FDP_EndStage (stage_id)
8:          Push the iteration to the in-queue of its next stage
```

Figure 7. Modified worker loop (additions/modifications are shown in bold)

Figure 7 shows how the code of the worker loop is modified to interface with the FDP library. The four function calls are inserted as follows. `FDP_Init` is called before the worker loop begins. Inside the loop the thread calls `FDP_GetNextStage` to get the ID of the next stage to process. The worker thread then pops an entry from the in-queue of the chosen stage. Before executing the computation in stage, it calls the instrumentation routine `FDP_BeginStage`. It then runs the computation and after the computation it calls the

instrumentation function `FDP_EndStage`. It then pushes the iteration to the in-queue of the next iteration.

3.7. Overheads

FDP is a pure software mechanism and does not require *any* changes to the hardware. FDP only incurs minor latency and software storage overhead. The latency overhead is incurred due to instrumentation and execution of the optimization phases. These overheads are significantly reduced because we only instrument 0.7% (1/128) iterations. The software storage overhead comprises the storage required for the current core-to-stage allocation, the list of previously tried core-to-stage allocations, the table to store execution latencies of each stage, and counters to support sampling. The total storage overhead is less than 4KB in a system with 16 cores and 16 stages. Note that this storage is allocated in the global memory and does not require separate hardware support.

4. Experimental Methodology

4.1. Configuration

We conduct our experiments on two real machines. Our baseline system is a Core2Quad SMP that contains 2 Xeon Chips of four cores each. To show scalability of our technique, we also conduct experiments with an AMD Barcelona SMP machine with four Quad-core chips (results for this machine will be reported in Section 6.4). Configuration details for both machines are shown in Table 2. Each system has sufficient memory to accommodate the working set of each of the workloads used in our study.

Table 2. System Configuration

Name	Core2Quad (Baseline)	Barcelona
System	8-cores, 2 Intel Xeon Core2Quad packages	16-cores, 4 AMD Barcelona packages
Frequency	2 GHz	2.2 GHz
L1 cache	32 KB Private	32 KB Private
L2 cache	Shared; 6MB/2-cores	Private; 512KB/core
L3 cache	None	Shared; 8MB/4-cores
DRAM	8 GB	16 GB
OS	Linux CentOS 5	Linux CentOS 5

4.2. Workloads

We use 9 workloads from various domains in our evaluation (including 2 from PARSEC benchmark suite [2][4]). Table 3 describes each workload and its input set. `MCarlo`, `BScholes`, `mtwister`, and `pagemine` were modified from original code to execute in pipeline fashion.

4.3. Measurements

We run all benchmarks to completion and measure the overall execution time of each workload using the GNU time utility. To measure the fine-grained timings, such as, spent inside a particular section of a program, we use the read timestamp-counter instruction (`rdtsc`). We compute the average number of active cores by counting the number of cores that are active at a given time and averaging this value over the entire execution

[4]The remaining PARSEC workloads are data-parallel (not pipelined) and FDP does not increase or decrease their performance

time. We run each experiment multiple times and use the average to reduce the effect of OS interference.

Table 3. Workload characteristics.

Workload	Description (No. of pipeline stages)	Input
MCarlo	MonteCarlo simulation of stock options [17] **(6)**	N=400K
compress	File compression using bzip2 algorithm [10] **(5)**	4MB text file
BScholes	BlackScholes Financial Kernel [17] **(6)**	1M opts
pagemine	Derived from rsearchk[14] and computes a histogram **(7)**	1M pages
image	Coverts an RGB image to gray-scale **(5)**	100M pixels
mtwister	Mersenne-Twister PRNG [17] **(5)**	path=200M
rank	Rank strings based on their similarity to an input string **(3)**	800K strings
ferret	Content based similarity search from PARSEC suite[2] **(8)**	simlarge
dedup	Data stream compression using deduplication algorithm from PARSEC suite[2] **(7)**	simlarge

5. Case Studies

FDP optimizes performance as well as power for pipelined workloads at runtime. We now show the working of FDP on both scalable and non-scalable workloads with the help of in-depth case studies that provide insights on how FDP optimizes execution. Detailed results and analysis for all workloads will be provided in Section 6.

5.1. Scalable Workload: Compress

The workload `compress` implements a parallel pipelined bzip2 compression algorithm. It takes a file as input, compresses it, and writes the output to a file. To increase concurrency, it divides the input file into equal size blocks and compresses them independently. It allocates the storage for the compressed and uncompressed data, reads a block from the file, compresses the block, re-order any work quanta which may have become out of order, writes the compressed block to the output file, and deallocates the buffers. Figure 3 shows the pipeline of `compress`. Each iteration in `compress` has 5 stages(S1-S5). Each stage can execute concurrently on separate cores, thereby improving performance.

Table 4 shows the throughput of each stage when each stage is allocated one core (the allocation 1-1-1-1-1). The throughput of stage S3, which compresses the block, is significantly lower than the other stages. Thus, the overall performance is dominated by S3 (the LIMITER stage). Table 4 also shows the throughput when one of the stage receives four cores and all other receive one core. For example, with the 4-1-1-1-1 allocation S1 receives four cores and all other stages get one core. Threads in S1 allocate buffers in the shared heap and contend for the memory allocator, thereby loosing concurrency, hence throughput of S1 improves by only 2.4x with 4x the cores. Whereas, when 4 cores are given to Stage S3, its throughput improves almost linearly by 3.9x because S3 compresses independent blocks without requiring any thread communication.

Table 4. Throughput of different stages as core allocation is varied. Throughput is measured as iterations/1M cycles.

Core Alloc.	S1	S2	S3	S4	S5	Exec. Time
1-1-1-1-1	284	49	0.4	34	8K	55 sec.
4-1-1-1-1	698	44	0.4	33	6K	55 sec.
1-4-1-1-1	294	172	0.4	35	7K	55 sec.
1-1-4-1-1	304	52	**1.5**	37	7K	**14 sec.**
1-1-1-4-1	279	49	0.4	135	8K	55 sec.
1-1-1-1-4	282	51	0.4	33	31K	55 sec.

Table 4 also shows the overall execution time with different core allocations. As S3 is the LIMITER stage, increasing the number of cores for other stages does not help reduce the overall execution time. However, when S3 receives more cores, the throughput of S3 increases by 3.9x and overall execution time reduces form 55 seconds to 14 seconds (a speedup of 3.9x). Therefore, to improve performance more execution resources must be invested in the LIMITER stage.

We modify the source code of `compress` to include library calls to FDP. FDP measures the throughput of each stage at runtime and regulates the core-to-stage allocation to maximize performance and power-efficiency. Figure 8 shows the overall throughput as FDP adjusts the core-to-stage allocation.

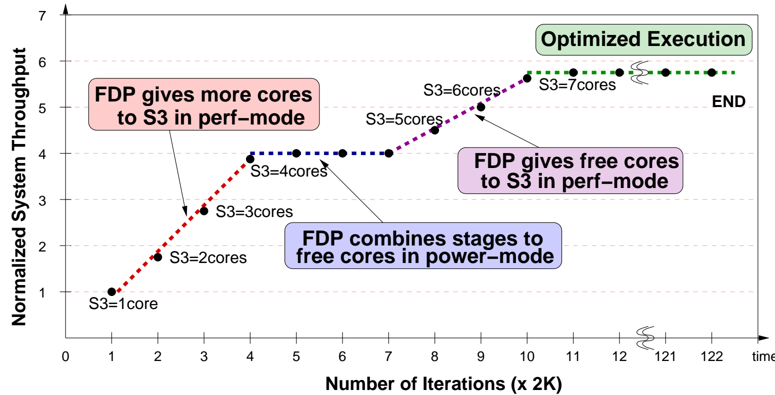

Figure 8. Overall throughput of `compress` as FDP adjusts core-to-stage allocation

FDP initially allocates one core to each stage. As execution continues, FDP trains and identifies S3 to be the LIMITER stage. To improve performance FDP increases the number of cores allocated to S3, until it runs out of cores. For our 8-core system, this happens when S3 is allocated 4 cores, and the remaining 4 cores are allocated one each to S1, S2, S4, and S5. After it runs out of cores, FDP begins to operate in power-optimization mode. In the first invocation of this mode, the stages with the highest throughput, S1 and S5, are combined to execute on a single core, thereby freeing one core. In the next invocation, FDP combines S1 and S5 with S2 which frees up another core. FDP continues this until all four stages S1, S2, S4, and S5 get combined to execute on a single core. With no opportunity left to reduce power, FDP switches back to performance optimization mode. FDP again identifies S3 as the LIMITER and allocates the 3 free cores to S3. Thus, 7 out of the 8 cores are allocated to S3, and a single core is shared among all other stages. FDP converges in 10 invocations and executes the workload in 9.7 seconds, which is much lower than with the static-best integer allocation (1-1-4-1-1) that requires 14 seconds.

5.2. Non-Scalable Workload: Rank

The `rank` program ranks a list of strings based on their similarity to an input string. It returns the top N closest matches (N is 128 in our experiments). Figure 9 shows the pipelined implementation for `rank`. Each iteration is divided into 3 stages. The first stage (S1) reads the next string to be processed. The second stage (S2) performs the string comparison, and the final stage (S3) inserts the similarity metric in a sorted heap, and

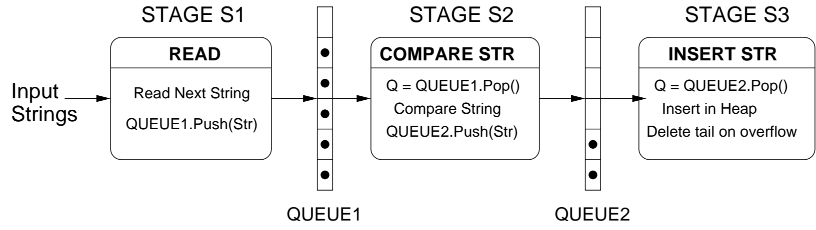

Figure 9. Pipeline for matching a stream of strings with a given string

removes the smallest element from the heap (except when heap size is less than N). At the end of the execution, the sorted heap contains the top N closest matches.

Table 5 shows the throughput of system when each stage is allocated one core (1-1-1). The throughput of S2, which performs the string comparison, is significantly lower than the other stages in the pipeline. As S2 is the LIMITER, allocating more cores to S2 is likely to improve overall performance. The next three rows in the table shows the throughput when one of the stage receives 4 cores and the other stages get one core. With the increased core count, S1 and S3 show a speedup of 2.5x and 1.3x, respectively. However, as these stages are not the LIMITER, the overall execution time does not decrease.

Table 5. Throughput of different stages as core allocation is varied (measured as iterations/1M cycles).

Core Alloc.	S1	S2	S3	Exec. Time
1-1-1	1116	142	236	17 sec
4-1-1	2523	118	258	19 sec
1-4-1	1005	558	278	13.2 sec
1-1-4	900	117	290	19.2 sec
1-4-2	930	368	285	14.6 sec
1-2-1	1028	274	268	13 sec

When S2 is allocated 4 cores, it shows the speedup of approximately 4x. This is because all cores in S2 work independently without requiring communication. Unfortunately, the overall execution time reduces only by 27%. This is because as S2 scales, its throughput surpasses the throughput of S3. Thus, S3 becomes the LIMITER. Once S3 becomes the LIMITER, the overall execution time is dominated by S3, making the improvements of S2 ineffective on the overall speedup.

As S3 is the LIMITER, we expect to improve overall performance by increasing cores allocated to S3. The table also shows the throughput when additional cores are allocated to S3 (1-4-2). The access to the shared linked data-structure in S3 is protected by a critical section, hence this stage is not scalable and overall performance reduces as the number of cores is increased due to contention for shared data. Thus, increasing core counts for S3 does not help improve performance while consuming increased power.

We modify the source code of `rank` to include library calls to FDP. Figure 10 shows the overall throughput and active cores as FDP adjusts the core-to-stage allocation. With the information obtained during training, FDP identifies S2 as the LIMITER stage, and allocates it one extra core (1-2-1). In the next invocation, it identifies S3 as the LIMITER stage, and increases the core count allocated to S3 (1-2-2). However, as S3 does not scale,

FDP withdraws the extra core given to S3, and switches to power-optimization mode. In power-optimization mode, FDP saves power by executing S1 on one of the cores allocated to S2. Thus, the final allocation is S1+S2 on one core, S2 on another core, and S3 on the third core. After this, there are no opportunities left in the pipeline to save power or improve performance, and execution continues on 3 cores completing in 13 seconds (similar to best-static allocation 1-2-1, but with fewer cores).

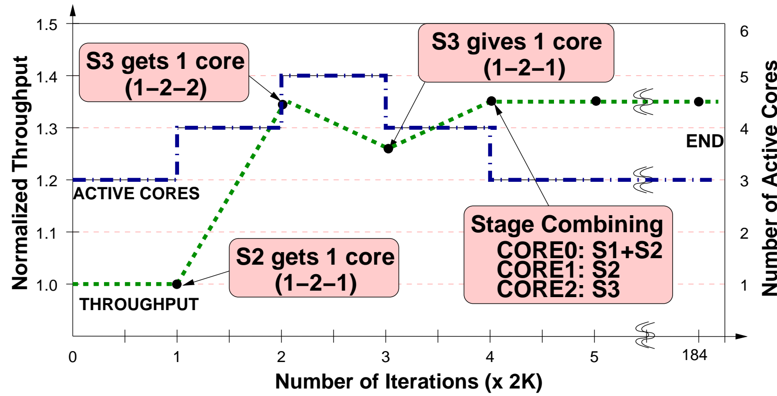

Figure 10. Overall throughput and active cores of `rank` as FDP adjusts core-to-stage allocation

6. Results

We evaluate FDP in terms of performance, power consumption, and robustness. We compare FDP with three core-to-stage allocation schemes. First, the *One Core Per Stage (1CorePS)* scheme which allocates one core to each stage. Second, the *Proportional Core Allocation (Prop)* scheme which allocates cores to stages based on their relative execution rates. Prop runs the application once with 1CorePS and calculates the throughput of each stage. The cores are then allocated in inverse proportion to the throughput of each stage, thus giving more cores to slower stages and vice versa. Third, the *Profile-Based* scheme which allocates cores using static profiling. The Profile-Based scheme runs the program for all possible allocations which assign an *integer* number of cores to each stage and chooses the allocation which minimizes execution time. Note that while the absolute best profile algorithm can try even non-integer allocations by allowing stages to share cores, the number of combinations with such an approach quickly approaches into millions, which makes it impractical for us to quantitatively evaluate such a scheme for this paper.

6.1. Performance

Figure 11 shows the speed-up when the workloads are executed with the core-to-stage allocation using 1CorePS, Prop, FDP, and Profile-Based. The speedup is relative to execution time with a single core system [5]. The bar labeled *Gmean* is the geometric mean over all workloads. The 1CorePS scheme provides only a marginal improvement, providing minor speedup increase on four out of seven workloads. On the contrary, a Profile-Based allocation significantly improves performance for all workloads, providing an average speedup of 2.86x. However, Profile-Based requires impractical searching through all possible integer allocations. Prop

[5]We run the sequential version without any overheads of multi-threading.

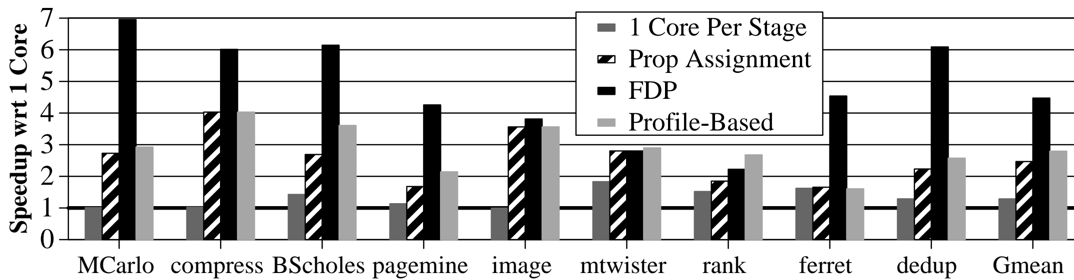

Figure 11. Speedup with different core-to-stage allocation schemes.

avoids this brute force searching and gets an improvement similar to Profile-Based by providing an average speedup of 2.7x. FDP outperforms or is similar to the comparative schemes on all workloads. `MCarlo` gets near optimal speedup of 7x with FDP because it contains a scalable LIMITER stage and FDP combines all other stages. The workload `rank` has a stage that is not scalable, hence the limited performance improvement with all schemes. FDP provides an average speedup of 4.3x. Note, that this significant improvement in performance comes without any reliance on profile information which is required for both Prop and Profile-Based.

6.2. Number of Active Cores

FDP tries to increase performance by taking core resources from faster stages and reallocating it to slower stages. When the slowest stage no longer scales with additional cores, the spare cores can be turned off or used for other applications. Figure 12 shows the average number of active cores during the execution of the program for 1CorePS, FDP, and Prop/Profile-Based. Both Prop and Profile-Based allocates all the cores in the system, therefore they are shown with the same bar. The bar labeled *Amean* denotes the arithmetic mean over all the workloads.

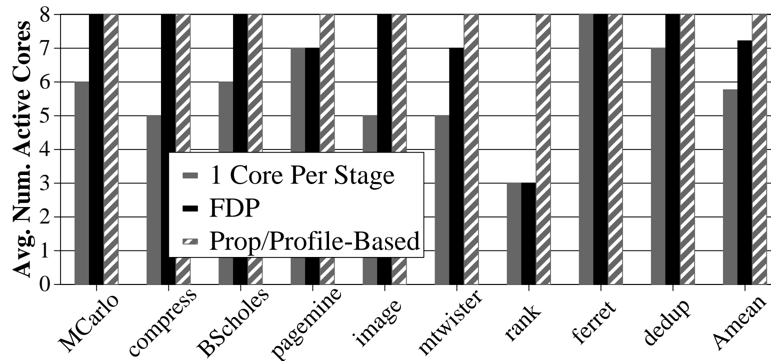

Figure 12. Average number of active cores for different core allocation schemes.

The number of active cores with the 1CorePS is equal to the number of pipeline stages, which has an average of 5.2 cores. The Prop and Profile-Based schemes use 8 cores. For `Pagemine` and `mtwister`, the performance saturates at 7 cores, so FDP does not use one of the cores in the system. For the workload `rank`, the non-scalable stage means that five out of the eight cores can be turned off. Thus, FDP is not only a performance enhancing technique but also helps with reducing the power consumed by cores when it is not possible to improve performance with more cores. On average, FDP consumes only 7 cores even though it has one and

a half times the speedup of the Profile-Based scheme. This means for the same number of active cores, FDP consumes two-thirds the energy as the Profile-Based scheme and has a much reduced energy-delay product.

6.3. Robustness to Input Set

The best core-to-stage allocation can vary with the input set. Therefore, the decisions based on profile information of one input set may not provide improvements on other input set. To explain this phenomenon, we conduct experiments for the `compress` workload with two additional input sets that are hard to compress. We call these workloads `compress-2` and `compress-3`. The LIMITER stage S3 for `compress-2` (80K cycles) and for `compress-3` (140K cycles) is much smaller than the one used in our studies (2.2M cycles). The non-scalable stage that writes to the output file remains close to 80K cycles in all cases. Thus, the `compress` workload has limited scalability for the newly added input sets.

Figure 13 shows the speedup for the two workloads with 1CorePS, Prop, FDP and Profile-Based. Both Prop and Profile-Based use the decisions made in our original `compress` workload. These decisions in fact result in worse performance than 1CorePS for `compress-2`, because they allocate more cores to the non-scalable stage which results in increased contention. FDP, on the other hand, does not rely on any profile information and allocates only one-core to the non-scalable stage. It allocates two cores to S3 for `compress-2` and 3 cores to S3 for `compress-3`. The runtime adaptation allows FDP to outperform all comparative schemes on all the input sets.

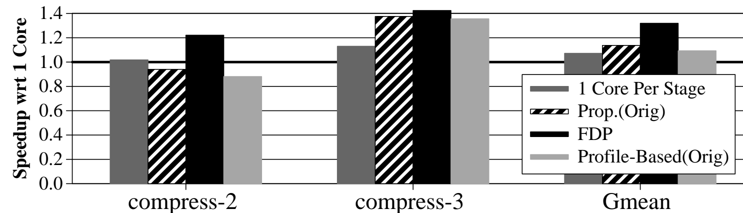

Figure 13. Robustness to variations in input set.

6.4. Scalability to Larger Systems

We use an 8-core machine as our baseline for evaluations. In this section, we analyze the robustness and scalability of FDP to larger systems, using a 16-core AMD Barcelona machine. We do not show results for 1CorePS as they are similar to the 8-core system (all workloads have fewer than 8 stages). Furthermore, a 16-core machine can be allocated to a 6-7 stage pipeline in several thousand ways, which makes evaluating Profile-Based impractical.

Figure 14 shows the speedup of Prop and FDP compared to a single core on the Barcelona machine. FDP improves performance of *all* workloads compared to Prop. Most notably, in `image`, FDP obtains almost twice the improvement of Prop. The scalable part of `image`, which transforms blocks of the image from colored to gray scale, continues to scale until 6 cores. The other parts, reading and writing from the file, do not scale. Prop allocates cores to each stage proportionally assuming equal scaling. However, the cores allocated to non-

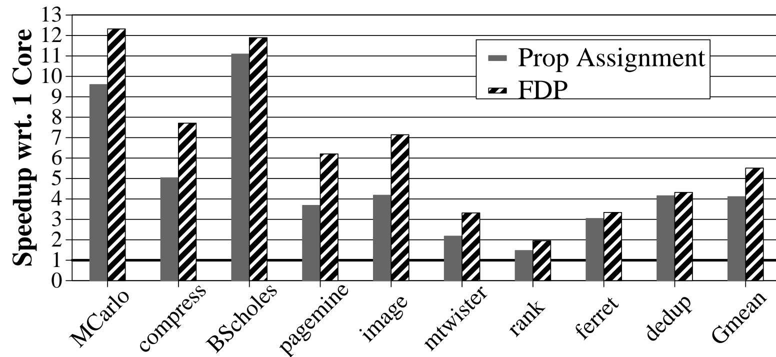

Figure 14. FDP's performance on 16-core Barcelona.

scalable parts do not contribute to performance. FDP avoids such futile allocations. On average, FDP provides a speedup of 6.13x compared to 4.3x with Prop.

As the number of cores increases, the performance of some of the workloads starts to saturate. Under such scenarios, there is no room to improve performance but there is a lot of potential to save power. Figure 15 shows the average number of active cores during the workload execution with FDP and Prop. Since Prop allocates all cores, the average for Prop is 16. When cores do not contribute to performance FDP can deallocate them, thereby saving power. For example, `pagemine` contains four stages in the pipeline that do not scale because of critical sections. FDP allocates 7 cores to the scalable stage, 1 core each to the non-scalable stages, and 1 more core to the input stage. The remaining four cores remain unallocated. On average, FDP has 11.5 cores active, which means a core power reduction of more than 25%. Thus FDP not only improves overall performance significantly but can also save power.

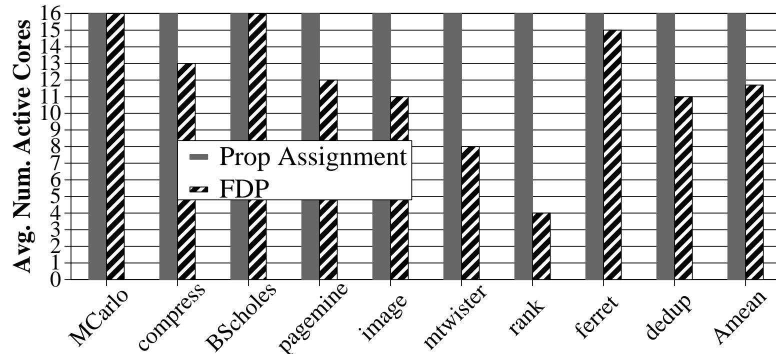

Figure 15. FDP's power on 16-core Barcelona.

If all cores were active, then the energy consumed by FDP would be 30% less compared to Prop (measured by relative execution time). Given that FDP uses 25% fewer cores than Prop, FDP consumes less than half the energy consumed by Prop. Thus, FDP is an energy-efficient high-performance framework for implementing pipelined programs.

7. FDP in Workloads with Work Sharing

Some parallel applications are implemented using the *Work Sharing* model instead of the pipeline model. Unlike the pipeline model, which sub-divides the work into stages, work sharing treats each iteration of the

work as a single unit of execution. In fact, work sharing can thus be viewed as a special case of pipelining, consisting of only one pipeline stage where all worker threads are assigned to that stage to execute identical pieces of execution. FDP can also be used to improve the performance of workloads implemented with the work sharing model. In such workloads, FDP treats the execution as consisting of a single stage, and chooses the number of threads which leads to maximum performance with the minimum number of cores.

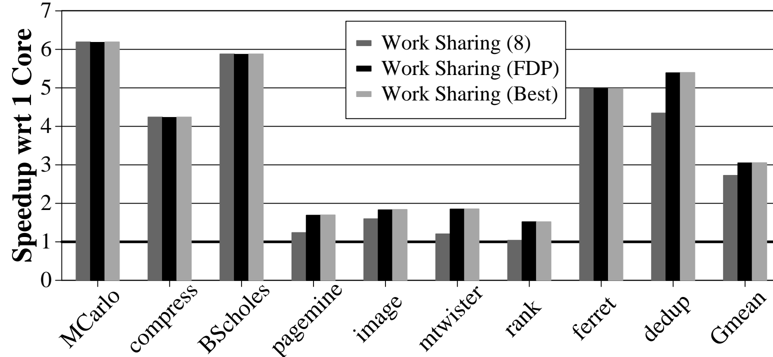

Figure 16. Comparison of FDP with work sharing.

A pipelined workload can be converted to a work sharing workload by forcing all stages of each iteration to run on the same core. Using this methodology, we converted the benchmarks used in our study to use work sharing and analyze the effectiveness of FDP for workloads implemented in work sharing.

Figure 16 shows the speedup with Work Sharing (with 8 threads), Work Sharing with FDP, and Work Sharing (Best). Work Sharing (Best) is an optimal scheme which tries all possible number of threads from 1-8 and picks the best performing configuration for each workload. In non-scalable workloads, where increasing the number of threads does not increase performance, Work Sharing (Best) has significantly higher (13-50%) performance than Work Sharing (8). For example, the workload `pagemine` has a long critical section. Performance of `pagemine` saturates at fours threads. Assigning it more than four threads increases critical section contention, which reduces performance and wastes power. Work Sharing (Best) chooses four threads for `pagemine` which leads to higher performance. Note that Work Sharing (FDP) performs the same as Work Sharing (Best). In fact, Work Sharing (FDP) is within 1% of Work Sharing (Best) in all workloads. Thus, FDP can effectively choose the best number of threads for work sharing workloads. FDP provides a speedup of 3.04x which is significantly higher than the 2.72x speedup of work sharing without FDP.

8. Related Work

With CMPs becoming the de-facto general purpose architecture, the emphasis on writing efficient and robust parallel programs has increased significantly. Several studies [8, 5, 2] have discussed the importance of using pipelined parallelism on CMP platforms. FDP provides automatic runtime tuning of core-to-stage allocation for this important paradigm and obtains improved performance and power-efficiency.

Recently Hormati et al. proposed the Flextream compilation framework [9] which can dynamically recom-

pile pipelined applications to adapt to the changes in the execution environment, e.g., changes in the number of cores assigned to an application. While FDP can also adapt to changes in the execution environment, its main goal is to maximize the performance of a single application. Flextream and FDP fundamentally differ for three reasons. First, unlike FDP, Flextream assumes that all stages are scalable and thus allocates cores based on the relative demands of each stage. This can reduce performance and waste power when a stage does not scale (see Section 5.2). Second, Flextream requires dynamic recompilation which restricts it to languages which support that feature, e.g., JAVA and C-sharp. In contrast, FDP is a library which can be used with any language. Third, Flextream cannot be used to choose the number of threads in work sharing programs because it will assume that the workload scales and allocate it all available cores. FDP, on the other hand, chooses the best number of threads taking scalability into account (see Section 7).

Other proposals in the operating system and web server domains have implemented feedback directed cores-to-work allocation [21, 19]. However, they make several domain-specific assumptions which makes their scheme applicable only to those domains, and less general than FDP.

The core-to-stage allocation can also be done statically using profile information. The brute force search for finding the best mapping can be avoided by using analytical models. Recently Navarro et al. [15, 16, 13, 6] proposed an analytic model for understanding and optimizing parallel pipelines. While such models can help programmers design a pipeline, they are static and do not adapt to changes in input set and machine configuration. In contrast, FDP relieves the programmer from obtaining representative profile information for each input set and machine configuration and does automatic tuning using runtime information.

Languages and languages extensions [7, 4, 10, 11] can help with simplifying the development of pipelined programs. Raman et al. [18] propose to automatically identify pipeline parallelism in a program using intelligent compiler and programming techniques. Our work is orthogonal to their work in that our proposal optimizes at run-time an already written pipelined program.

Pipeline parallelism is also used in databases [1] where each database transaction is split into stages which can be run on multiple cores. Their work can also use FDP to choose the best core-to-stage allocation.

Although FDP primarily targets programs written in pipelined model, it can also improve performance and power of non-pipelined programs such as those amenable to work-sharing. Several schemes [3, 20, 12] tune thread-to-core mapping of data-parallel workloads implemented using work-sharing paradigm. However, these proposals are not applicable to pipelined programs. To the best of our knowledge, FDP is the only comprehensive framework that improves performance and power-efficiency of both pipelined workloads as well as data parallel workloads.

9. Conclusion

Pipeline parallelism is a common technique to improve performance of a single application using multiple cores. The potential of pipelining is not fully utilized unless all the stages are balanced in terms of execution rate, which can be controlled by adjusting the core-to-stage allocation. Unfortunately, it is challenging for the programmer to decide the core-to-stage allocation because the best allocation depends on the input set and machine configuration. Furthermore, a brute-force search for the best configuration is impractical and can require up to a million runs. A dynamic mechanism that can learn the best core-to-stage allocation using runtime information can overcome these limitations. This paper proposes *Feedback-Directed Pipelining (FDP)*, a framework to choose the best core-to-stage allocation at runtime and makes the following contributions:

1. It proposes a practical framework to monitor execution time of each stage at runtime in a cost-effective manner. This information can be used to identify the slowest stage and the fastest stage in the pipeline.
2. The proposed FDP framework uses the runtime information to learn the best core-to-stage allocation, using a hill-climbing algorithm. The slowest stage is given resources until either there are no more spare cores or the performance of the stage saturates.
3. When performance saturates, FDP tries to free cores by combining the faster stages to run on one core. The core thus freed can be used to improve performance or save power.

We evaluate FDP on an 8-core Core2Quad SMP, using 9 multi-threaded workloads. FDP provides an average speedup of 4.3x (compared to 2.8x with profile based allocation) while at the same time reducing the number of active cores by 12.5%. We also evaluate FDP on a 16-core Barcelona system and show that FDP continues to provide significant performance and power benefits. FDP has a simple interface with only four function calls, and requires minimal programmer intervention. We intend to make the FDP library available for public use.

10. Future Work

FDP is a runtime mechanism which can detect performance limiters of an application, and then invest available resources to accelerate the limiters. We envision FDP to have a major role in future systems: FDP can be used in systems with heterogeneous cores. In systems where cores differ in performance or functionality, FDP can choose for each stage the core best suited to run it. FDP can also be extended to other execution paradigms such as using FDP for task-scheduling in task-parallel workloads.

References

[1] S. Anastassia and A. Ailamaki. Stageddb: Designing database servers for modern hardware. In *In IEEE Data*, 2005.

[2] C. Bienia et al. The parsec benchmark suite: characterization and architectural implications. In *PACT 2008.*

[3] M. F. Curtis-Maury. *Improving the Efficiency of Parallel Applications on Multithreaded and Multicore Systems*. PhD thesis.

[4] A. Das et al. Compiling for stream processing. In *PACT '06*, 2006.

[5] J. Giacomoni et al. Toward a toolchain for pipelineparallel programming on cmps. *Workshop on Software Tools for Multi-Core Systems*, 2007.

[6] D. González et al. Towards the automatic optimal mapping of pipeline algorithms. *Parallel Comput.*, 2003.

[7] M. I. Gorden et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII*, 2006.

[8] J. Gummaraju et al. Streamware: programming general-purpose multicore processors using streams. *SIGARCH Comput. Archit. News*, 2008.

[9] A. H. Hormati et al. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT '09*, 2009.

[10] Intel. Source code for Intel threading building blocks. http://www.threadingbuildingblocks.org/, 2009.

[11] J. R. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, 2002*.

[12] J. Li et al. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *HPCA*, 2006.

[13] W.-K. Liao. Performance evaluation of a parallel pipeline computational model for space-time adaptive processing. *J. Supercomput.*, 2005.

[14] R. Narayanan et al. MineBench: A Benchmark Suite for Data Mining Workloads. In *IISWC*, 2006.

[15] A. Navarro et al. Analytical modeling of pipeline parallelism. In *PACT'09*, 2009.

[16] A. Navarro et al. Load balancing using work-stealing for pipeline parallelism in emerging applications. In *ICS*, 2009.

[17] Nvidia. CUDA SDK Code Samples. http://developer.download.nvidia.com/compute/cuda/-sdk/website/samples.html, 2007.

[18] E. Raman. Parallel-stage decoupled software pipelining. In *CGO '08*, 2008.

[19] D. C. Steere et al. A feedback-driven proportion allocator for real-rate scheduling. In *OSDI'99*.

[20] M. Suleman et al. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *ASPLOS XIII*, 2008.

[21] M. Welsh et al. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01).*

Appendix: Pseudo-code for FDP

```
Global variables
mode // power or performance mode
LastTp // Last throughput of pipeline
CurrAlloc // Current core-to-stage allocation
LastAlloc // Previous allocation
TStart // Time at start of training
PAllocSet // Previously tried allocations set
TMeasured // 2D table like Figure 5
ICount // Iterations processed during Train

FDP_Init:
mode       =  performance-mode
LastTp     =  0
CurrAlloc  =  1 core-per-stage
LastAlloc  =  1 core-per-stage
TStart     =  cycle_count_register
PAllocSet  =  empty
TMeasured  =  All values equal 0
ICount     =  0

FDP_BeginStage(stage_id):
core_id = cpuid_register
if(stage_id is 0)
   ICount++
if(128th call of FDP_BeginStage)
  Tnow = cycle_count_register
   Record Tnow as start time of core_id, stage_id
  if(ICount is 2000 or (Tnow - TStart) > 100K)
     Call FDP

FDP_EndStage(stage_id):
core_id = cpuid_register
if(128th call of FDP_EndStage)
  Tnow = cycle_count_register
  TE = Tnow - start time of core_id, stage_id
  Update TMeasured[stage_id,core_id]
     with TE

FDP_GetNextStage:
Choose a stage in round-robin fashion from
  set of stages with non-empty in-queues

FDP:
NewTp //Local: New pipeline throughput
TElapsed //Local: Cycles in last train
TElapsed = cycle_count_register - TStart
NewTp = TElapsed/ICount
if(NewTp < LastTp)
   Exchange(CurrAlloc, LastAlloc)
   Toggle mode
else
   Call power- or performance-mode
   If NewAlloc belongs to PAllocSet
      Toggle mode
   else
      LastAlloc = CurrAlloc
      CurrAlloc = NewAlloc

PAllocSet.insert(CurrAlloc)
LastTp = NewTp
ICount = 0
TStart = cycle_count_register
TMeasured.reset()

performance-mode:
//Local variables
NewAlloc //The new allocation
LIMITER // the ID of the LIMITER stage

Compute throughput of all stages
LIMITER = stage with minimum throughput

if(spare cores are available)
   New Allocation = CurrentAllocation
      with one additional core for LIMITER
else
   mode = power-mode

power-mode:
Compute throughput of all stages
S1,S2 = two stages with least thoughput
NewAlloc = CurrentAlloc with two changes:
   Assign S2 to S1's core
   De-allocate S2'sallocation
```