Multitasking Workload Scheduling on Flexible Core Chip Multiprocessors

Divya P. Gulati Changkyu Kim Simha Sethumadhavan Stephen W. Keckler
Doug Burger

Department of Computer Sciences
The University of Texas at Austin
{cartel}@cs.utexas.edu

Abstract

While technology trends have ushered in the age of chip multiprocessors (CMP) and enabled designers to place an increasing number of cores on chip, a fundamental question is what size to make each core. Most current commercial designs are symmetric CMPs in which each core is identical and range from a relatively simple RISC pipeline to a large and complicated out-of-order x86 core. When the granularity of parallelism in the tasks matches the granularity of the processing cores, a CMP will be at its most efficient. To adjust the granularity of a core to the tasks running on it, recent research has proposed flexible-core chip multiprocessors, which typically consist of a number of small processing cores that can be aggregated to form larger logical processors. These architectures introduce a new resource allocation and scheduling problem which must determine how many logical processors should be configured, how powerful each processor should be, and where/when each task should run. This paper introduces and motivates this new scheduling problem, describes the challenges associated with it, and examines and evaluates several algorithms (amenable to implementation in an operating system) appropriate for such flexible-core CMPs. We also describe how scheduling for flexible-core architectures differs from scheduling for fixed multicore architectures, and compare the performance of flexible-core CMPs to both symmetric and asymmetric fixed-core CMPs.

1 Introduction

While technology trends have ushered in the age of chip multiprocessors (CMP) and enabled designers to place an increasing number of cores on chip, a fundamental question is what size to make each core. Most current commercial designs are symmetric CMPs (SCMPs) in which each core is identical and range from a relatively simple RISC pipeline to a large and complex out-of-order x86 core. However, the concurrency characteristics of programs to be run on a CMP demonstrate substantial diversity. For example, the amount of available instruction-level parallelism across different applications may vary widely [14]. Even the characteristics of a single program may vary during different phases of its execution [12]. Selecting the number of cores and the size of the cores at design time will result in inefficiencies when the characteristics of the workload do not match the fixed parameters of the system. An alternative to SCMPs are asymmetric chip multiprocessors (ACMPs) which typically comprise multiple processors of different sizes and granularities. Such a design allows individual applications or application phases to be mapped to the processor size best suited to it, resulting in better power efficiency, greater throughput, and better area efficiency (defined as throughput per unit area) than SCMPs. However, the composition of the ACMPs must still be determined at design time, leaving such multicore processors vulnerable to mismatches between the workload and the available hardware.

Recently proposed alternatives to fixed-core CMPs are a family of flexible-core chip multiprocessors (FCMPs) in which the number and granularity of the processors is determined at runtime through aggregation and configuration [7, 9, 13]. Such designs typically comprise small to moderately sized uniprocessor cores which can execute in parallel as a multitasking/parallel system or which can be aggregated together to form fewer but more powerful uniprocessor cores. The aggregation typically produces a core with higher issue width, a larger instruction window, and more level-1 instruction and data cache capacity. The flexibility of FCMPs provides the opportunity to tailor the hardware to the requirements of the tasks running on the system, or to co-optimize the software and the configuration of the underlying hardware. FCMPs offer a number of advantages over ACMPs, including the opportunity to map a wider range of workloads, simpler hardware implementation as all of the cores of an FCMP can be identical [7], and better tolerance to performance asymmetries resulting from the fixed but varying cores [1]. The flexibility in FCMPs also allows optimization of different metrics such as performance, power efficiency, and area efficiency. When combined with Dynamic Voltage Frequency Scaling (DVFS), the range of configuration possibilities can be manifold.

While providing flexibility is one challenge for FCMP architectures, managing the resources for a FCMP is a ma-

jor challenge for such a system. A scheduler and resource allocator must determine (1) how many logical processors to assemble from the cores, (2) how large each processor should be and whether the allocation should be symmetric or asymmetric, (3) what topology each logical core should take, (4) where each task should run, (5) under what circumstances should the assignment of tasks to processors change, and (6) the optimal way of migrating task state on reconfiguration. While ACMPs may require some aspects of (4) and (5), determining the configurations and assignments cooperatively is a new problem unique to FCMPs.

This paper introduces and motivates this novel scheduling problem, describes the challenges associated with it, and presents several operating-system amenable algorithms suited to FCMP resource allocation and scheduling. We explore the problem in an environment consisting of multiprogrammed single-threaded workloads with both fixed and dynamic workloads. The fixed workloads range from one to 16 tasks all of which are available at time zero; the dynamic workloads include a Poisson rate of task arrival which causes the number of running tasks to vary over time. We adapt scheduling algorithms from the multiprocessor scheduling literature to FCMPs and compare them to existing algorithms for symmetric and asymmetric CMPs. In our experiments, we focus solely on performance, for which we use response time, defined as the time elapsed between a task's arrival and departure, as the metric. For the sake of simplicity, we restrict this configurability of the architecture to the cores and leave an examination of DVFS for future work.

Section 2 describes in further detail the characteristics of a flexible-core CMP as well as the specific TFlex flexible-core architecture we use in this study [9]. Section 3 describes prior scheduling approaches for fixed core SCMPs and ACMPs, and the challenges common to scheduling for CMPs in general and for FCMPs in particular. Section 4 describes a set of our proposed scheduling algorithms, Section 5 discusses our experimental methodology, and Section 6 presents the results of our experiments.

2 Flexible-Core Architectures

Flexible-core architectures seek to provide adaptivity in the number and granularity of processors, enabling the system to efficiently execute both a large and a small number of threads. The basic approach is to aggregate a number of smaller identical processors to form larger logical processors. One example of a flexible core architecture is Core Fusion, which provides mechanisms to enable multiple out-of-order cores to be fused into a single more powerful core [7]. Federation is a similar solution, but instead federates multiple in-order cores to create an out-of-order processor [13]. While these approaches have the advantage of working with conventional instruction set architectures, the sequential execution model may hinder scaling the number of aggregated cores. Voltron applies a related approach to fuse multiple VLIW cores into a larger VLIW core [15]. Supporting this degree of flexibility requires physical distribution of different architectural structures including the register file, instruction window, L1 caches, and operand bypass network. In addition to the partitioning, various distributed control protocols are required to correctly implement instruction fetch, execute, commit, speculation recovery, and other processor actions.

In this paper, we use the TFlex Configurable Lightweight Processor (CLP) multicore architecture because of its ability to aggregate up to 32 cores into a single logical core [9]. Each CMP core is a simple dual issue out-of-order processor. Figure 1 (taken from [9]) shows a high-level floorplan of a TFlex processor. In addition to the grid of cores, TFlex contains 32 banks of level-2 cache arranged in a non-uniform (NUCA) architecture [8]. The figure also shows three of many possible configurations of a TFlex system. The configuration in Figure 1a has 32 1-core processors, Figure 1c has one 32-core processor, and Figure 1b has a diverse mix of processors. Throughout the rest of this paper we will be using a terminology of P-*N* to refer to a logical processor with *N* cores. Thus Figure 1b shows a configuration with two P-8s, two P-4s, and four P-2s. The rest of this section describes the basic architecture of TFlex, how cores are composed into larger processors, and considerations for reconfigurability.

ISA support: The scalability of flexible core architectures can be hindered by an execution model that requires a sequential instruction stream fetched from a common instruction store. The execution model also influences the ease of distributing different microarchitecture structures, such as the operand bypass bus, the register rename table, and load/store queues. To address these challenges, TFlex employs an Explicit Data Graph Execution (EDGE) ISA, which enables distributed instruction fetch and makes explicit the communication between different instructions [2].

EDGE ISAs are characterized by two properties. The first is block-atomic execution, in which control protocols for instruction fetch, completion, and commit operate on blocks (chunks of instructions containing up to 128 instructions on TFlex). This model of execution amortizes overheads like those of branch prediction and commit, making them tolerant to latency as would be seen in an composed processor. The second is that instructions explicitly encode the address of their consumers. This simplifies operand bypass hardware, which simply has to route the data produced by an instruction to the consumer, rather than broadcasting it on a bus. The instructions are interleaved across all cores composed as one processor, in a specific order, which helps to locate them using a point-to-point network. The inter-

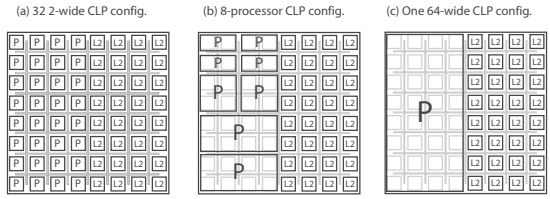


Figure 1: Three dynamically assigned CLP configurations.

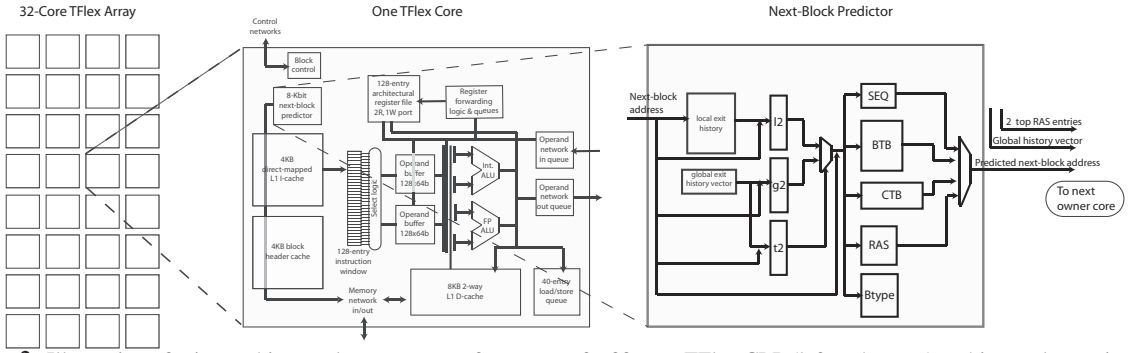


Figure 2: Illustration of microarchitectural components of one core of a 32-core TFlex CLP (left and center) and internal organization of the next-block predictor (right).

leaving changes when the core allocation changes.

Microarchitectural support: TFlex achieves full composability, which means that no structures are shared across cores. Figure 2 (from [9]) illustrates the various microarchitectural components of a TFlex core. When a core operates as its own dual-issue processor, all of the microarchitecture structures are local. When cores are aggregated, the logical instruction window, register file, instruction cache, data cache, and branch predictor are interleaved over the participating cores. These structures are addressed partitioned in the following manner: (1) *Block starting address* (equivalent of PC in conventional architectures) is used to partition the next-block predictor (branch predictor) structures and block tag structures. Each block is assigned an owner core based on the starting block address; ownership can rotate among the participating cores. (2) *Instruction IDs* (within blocks) are used to partition the instruction window and I-cache, which results in a block getting equally distributed across all participating cores. (3) *Data address* is used to partition L1 data cache and register files.

Since interleaving is controlled by bit-level hash functions, all logical processor sizes must consist of a power-of-two number of cores. All components of block execution, including branch prediction, instruction fetch, instruction execution, memory access, commit, and misspeculation recovery are distributed across the participating cores and are implemented using pipelined and distributed protocols. The protocols are realized using the control and data networks of the TFlex architecture. Additional details of the TFlex architecture are found in [9].

Processor reconfiguration: To be effective at adapting to a changing number and type of simultaneously executing programs, a flexible core architecture must (1) be able to isolate the tasks running on separate logical processors, and (2) be easily configured from one arrangement of logical processors to another. In TFlex, each core is allocated a *physical ID*, which for a 32-core system, ranges from 0 to 31. Within each logical processor, all participating cores are assigned a *logical ID*. To provide each task with its own exclusive processor, the hardware implements a virtualization layer that translates the logical core IDs to physical core IDs. Each participating core contains a configuration map which provides a mapping of its logical core IDs to physical core IDs so that the control networks implement the processor control protocols correctly. In addition, each core also knows the size of its configuration, which is used to compute the interleaving factors. Finally, cache lines in the L1 data caches maintain an address space ID (ASID) so that they do not have to be flushed on reconfiguration.

Reconfiguration of a logical processor requires three steps: (1) stopping the pipeline, (2) moving state from the old set of cores to the new set of cores according to the new interleaving, and (3) adjusting the configuration registers to reflect the new mapping of physical cores to logical processors. When stopping the pipeline, each I-cache must be invalidated because a block's tag is cached only at the owner core and the execution protocol requires the data blocks to be present in all of the other cores. Only the registers must be moved from the old configuration to the new one and interleaved according to the size of the new configuration. Because TFlex provides cache coherence at the L1/L2 interface, invalidating the L1 D-cache is not neces-

sary; the blocks will automatically be fetched on demand into the new cache banks. Ignoring the overhead associated with context switching into the operating system, we estimate the latency for a simple reconfiguration to require 500 cycles. Section 5 describes the modeling of reconfiguration latency in greater detail.

3 Scheduling for Fixed-Core CMPs

3.1 Symmetric CMPs

In a traditional multitasking parallel system, a scheduler must decide what tasks run and on what processors. Events triggering scheduling typically include task arrival, completion, and interrupt. If all resources are equivalent and all tasks are independent, scheduling is a straightforward process that can most trivially be implemented in a first-come, first-served algorithm. Periodic rescheduling can ensure a fair allocation of resources to tasks. In such systems, scheduling can be infrequent enough to be implemented in the operating system on millisecond timescales. Scheduling becomes more challenging when tasks are composed of multiple threads that interact with one another; gang scheduling is effective at ensuring that tasks that interact are run simultaneously [5]. For symmetric CMPs (SCMPs), we assume that tasks are independent of one another and use a simple first-come, first-served scheduler which runs each task to completion. We also assume a model that uses the operating system to perform the scheduling, but recognize that finer grained rescheduling and reconfiguration could benefit from more frequent and faster scheduling.

3.2 Asymmetric CMPs

The scheduling process becomes more challenging when the individual cores in a CMP have different characteristics as the scheduler must not only decide which tasks to run, but also which processor to run each task on. A good match between the characteristics of a task and the processor on which it runs will result in an efficient use of resources. Take, for example, a hypothetical two-core CMP with one core having twice the issue width of the other. Further assume a workload with two tasks: (1) a high-ILP task whose instruction throughput scales with the issue width, and (2) a low-ILP task whose instruction throughput is independent of issue width. In the best assignment, this CMP will achieve a throughput of $3\times N$ instructions per cycle, where N is the issue width of the narrower processor. In the worst assignment, the system will achieve a throughput of only $2\times N$.

While asymmetric CMPs are a relatively new phenomenon, Kumar et al. have examined a family of scheduling and resource allocation algorithms for them [10]. Their algorithms can be placed into three categories: static, random, and dynamic. The static algorithm assumes that information about how each task performs on each processor type is available a priori. This is then used to find an optimal assignment of tasks to processors. Any time a task departs or a new task arrives, the scheduler tries to find a new optimal assignment, and can use a number of optimization algorithms, including dynamic programming. The random algorithm simply finds a random assignment but ensures that more powerful cores get used before less powerful ones.

In contrast, the dynamic algorithms attempt to adapt to dynamic changes in the environment like task arrival/departure or task phase changes. Kumar et al. divide the algorithm into two phases: *sample* and *steady*. In the sampling phase, the scheduler tries various different assignments of tasks to processors to find the "best" one, by permuting the mapping of tasks to processors and measuring the effect on throughput. Then the best assignment or sample is maintained in the steady phase, which is much longer than the sampling phase. In the steady phase, throughput is monitored for deviations to determine when task reassignment might be necessary.

Sampling algorithms: Kumar et al. describe several sampling algorithms, but we limit the discussion here to their best performing one called *sample-sched*. In sample-sched the scheduler runs at most $4*n$ different assignments of tasks to processors, where n is the number of tasks in the system. This subset of the total possible configurations is chosen randomly except for a constraint that each task must be run at least once on the least powerful processor. The assignment with the best throughput is used for the next steady phase. Their algorithm uses *weighted speedup* (WS) as a metric of performance which is defined as the sum of individual speedups achieved by each task for a particular configuration. Individual speedup of a task on a given processor is computed as the ratio of its throughput on that processor to that on the least powerful processor.

Triggering a sampling phase: Kumar et al. found that triggering sampling periodically performs worse than a policy that triggers sampling only after a change in the workload or environment. Their best policy, called *bounded-global-trigger*, triggers sampling when the measured throughput changes by more than a certain threshold. In this policy the performance of each application is monitored during the steady phase and the absolute value of the percentage change in its IPC is calculated periodically. A sampling phase is triggered if the sum of the percentage IPC change of all tasks running exceeds a 100%. In order to guard against short phase behavior, this policy delays sampling until steady phase has run for a minimum threshold number of cycles. Likewise, when steady phase runs for a large number of cycles, exceeding a second threshold, the algorithm triggers a sampling phase. To this policy we add

the condition that sampling should be triggered if a task departs or a new one arrives and the lower threshold has been exceeded.

For our experiments, we chose to model the duration of the sampling interval as 50K cycles. We set the lower- and upper-bounds for the *bounded-global-trigger* at 1M and 5M cycles, respectively, meaning that the steady phase will run for at least 1M and no more than 5M cycles. All of these parameters are relatively small in order to accommodate the slow execution rate of our simulator. While these values are not nearly as large as those chosen by Kumar et al., their ratios are similar to those in the original study. This methodology will discover the smaller phases, while the larger phases will be beyond the measurement capability of the simulator.

4 Scheduling for Flexible-Core CMPs

Flexible-Core CMPs present a number of unique challenges to a scheduler. The resource allocation and scheduling problem has the following components.

(1) Determining the number and size of logical processors: Determining the arrangement of cores into logical processors is complicated by the sheer number of possible configurations. For example, a 32-core TFlex FCMP has 2279 unique configurations, where a configuration determines the number and size of the logical processors. One configuration of such a FCMP might include four logical processors: one P-16 (16 cores), one P-8 (8 cores), and two P-4s (four cores each). The number and size of the processors is influenced by the number of tasks that are ready to run, by the parallelism profiles of the tasks which may vary across phases of a task, and by the degree of contention for shared resources like the level-2 cache. Scheduling for a processor like TFlex requires finding a configuration that best suits the workload and assigning the active tasks to it.

(2) Determining the topology of each logical processor: Flexible-Core CMPs also expose a tradeoff in the shape of a logical processor. For example, although a 4-core processor could be arranged as a 2x2 or a 1x4, the 2x2 will generally have better performance as it minimizes communication distances. However, if only a 1x4 space is available, throughput will benefit from running a task on those four cores rather than leaving them idle.

(3) Where should each task run: This not only involves finding a vacant location for each task that has been allocated some cores, but doing it in a manner that minimizes shuffling already running tasks. Fragmentation can make this problem more challenging as shown in figure 3(a). The figure shows an assignment of cores to processors after three tasks terminate. The shaded regions show the vacated cores, that in this case, occupy discontiguous regions of the array. Even though collectively there are eight free cores available, assigning them to a new task in a contiguous manner would require shuffling already running tasks around. The problem of allocating cores in the free regions of a multiprocessor has been studied extensively in the domain of multiprogrammed parallel systems, an excellent survey of which is provided in [5].

(4) When should the configuration or assignment of tasks to processors change: One of the objectives of a scheduler is to minimize time spent in less than ideal configuration. This requires deciding under what circumstances should the configuration change. Significant events like task arrival, task departure, or substantial changes in the system performance can be used to consider reconfiguration. Issues of fairness and priority can complicate this problem further. Finally, a tradeoff exists between running in less than ideal configuration and the overhead of reconfiguration.

(5) State migration during reconfiguration: When a logical processor is reconfigured for a running task, its state must be migrated from the old mapping to the new mapping. As described in Section 2, the memory state can be migrated automatically, using built-in cache coherency mechanisms. Registers can be moved in one of two ways. The first method looks very much like a context switch as the registers from the reconfigured processors are stored to memory, the processors are reconfigured, and the register values are retrieved from memory. The second method minimizes the memory traffic by transmitting the register values directly from the old mappings to the new mappings. This second method requires an ordering to the reconfiguration of multiple processors to ensure that no state is lost during the register remapping. The operating system is a natural mechanism for performing the reconfiguration, but we also envision state machines being capable of recognizing when reconfiguration is desired and performing it without software intervention.

While the challenges for FCMP scheduling are numerous, we focus primarily on the problem of selecting the right size and number of the logical processors. To that end, we examine configuration scenarios that do not encounter the issues of fragmentation and shape. We classify the space of FCMP scheduling algorithms into three categories: static, profiling-based, and dynamic.

Static algorithms: We define the class of static algorithms as those that ignore the characteristics of individual tasks, but may account for the number of running tasks. One simple algorithm is to divide the number of cores into logical processors of equal size and assign each one to a task. We term this algorithm **EQUI** to reflect the equal allocation of cores. Thus each task receives the same-sized processor regardless of its concurrency and resource demands. Each time the number of tasks changes due to arrival or departure, the algorithm re-calculates the number of cores and changes the allocation if this number has changed. For example in a 32-core system with two tasks, each will get 16

cores. Such a distribution can result in idle cores if the total number of cores is not evenly divisible by the number of tasks. A slightly different version of this algorithm has been used in a number of studies on scheduling for parallel systems [5].

Profile-based algorithms: This class of algorithms assumes that some information about the characteristics of each application is available to the scheduler at its arrival time. While this information could be obtained through a priori or on-line profiling, we anticipate that compile-time analysis could also provide hints about a task's requirements.

One simple algorithm allocates each task its ideal number of cores on a first-come, first-served (FCFS) basis. If the ideal core count for the task at the head of the FCFS queue is available, the task is mapped onto those cores. Otherwise, the task waits in the queue. If the task at the head of the queue must wait, the scheduler could choose to maintain the FCFS ordering by forcing all other tasks to wait. Another option is to *backfill* by finding later arriving tasks that have fewer resource requirements to fill in the gaps. While backfilling has been extensively studied in multiprogrammed parallel systems [4], we have not yet explored this algorithm.

We call the profile-based algorithm that we implement and examine in this paper, **Profile**. This algorithm assumes that information about how a task's weighted speedup varies with core count (cores-to-ws function) is available at arrival time. Using this it finds an allocation of cores that maximizes the weighted speedup of the system as a whole. This can be stated as the following optimization problem:

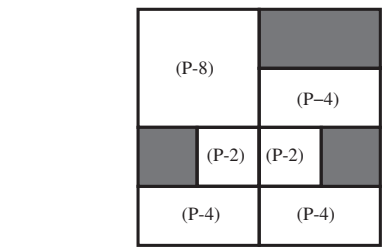
Maximize $\sum_{i=1}^{t} f_i(c_i)$, given the constraint $\sum_{i=1}^{t} c_i <= N$

where f_i is the cores-to-ws function and c_i is the number of cores allocated for task i, t is the total number of tasks, and N is the total number of cores in the system.

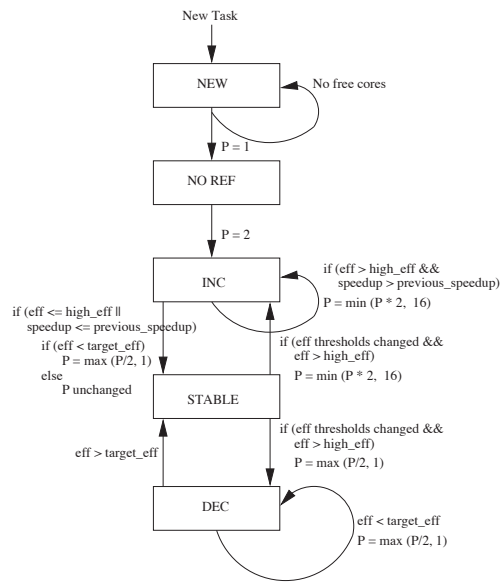
We solve this optimization problem by using an optimal dynamic programming algorithm with $O(tN^2)$ complexity [6]. Anytime a task departs or a new one arrives, the scheduler runs the dynamic programming algorithm to find the new optimal allocation.

Dynamic algorithms: While Profile exploits knowledge of the characteristics of an individual task, it relies on this knowledge being available a priori. Such profiling information may not always be available or may be inaccurate due to differences in the profiled and real execution of the program. The obvious alternative is to acquire this information online.

We call our dynamic algorithm **PDPA**, which stands for Performance Driven Processor Allocation and is adapted from [3]. Our modified version of the algorithm allocates cores based on how efficiently a task is executing, where efficiency is defined as (Speedup_over_1_core) / (Number_of_cores_currently_allocated). If a task achieves an efficiency higher than a predefined threshold called *high_eff*, the task can acquire more cores. Similarly, if it achieves an efficiency lower than another predefined threshold called *target_eff*, cores are taken away. Otherwise the allocation is maintained. Note that the thresholds are chosen such that target_eff $<$ high_eff.



(a) Fragmentation on a flexible-core CMP



(b) PDPA state diagram

Figure 3

The scheduler implements a state machine shown in figure 3(b) with each task being in one of its states. It evaluates tasks periodically whenever the timer interrupt goes off. The next state and the new allocation of a task is determined based on how it performed in the just completed interval, and is described below.

NEW state: Newly arriving tasks are initially placed in the NEW state. If a task was in this state in the last interval, the scheduler allocates it one core (shown as $P = 1$ in the state diagram), if one is available, and changes its state to NOREF. The task stays in the NEW state if no free core is available.

NOREF state: The purpose of this state is to acquire a baseline reference for a task by executing it on a single core. If a task was in this state in the last interval, its reference has been acquired. Therefore, it is placed in the INC state with an allocation of two cores.

INC state: In this state, the scheduler tries to determine how well the task utilized the increased allocation and whether it can benefit by a further increase. If the task achieved an efficiency greater than the high_eff threshold *and* if it achieved a speedup greater than it did in the previous interval (the interval prior to the one being evaluated), the allocation of the task is doubled and its state maintained as INC. However, if this is not true, the task's state is changed to STABLE. Furthermore, in the latter case if the task's efficiency dropped below the target_eff threshold, its allocation is halved. The allocation never exceeds 16 cores and never becomes zero until the task terminates.

DEC state: This state is the analog of the INC state. If a task's efficiency drops below the target_eff, its allocation is halved and its state maintained as DEC. Otherwise, its state is changed to STABLE.

STABLE state: When a task reaches the STABLE state, it is assumed to have an optimal allocation, and is therefore maintained there. However, this may no longer be desirable if the load on the system changes significantly. For example, if the system transitions from a moderate load to a light load, it may be better to allocate more cores to the existing tasks. In order to accomplish this the scheduler maintains different values of the efficiency thresholds according to the load on the system, and re-evaluate tasks in the STABLE state whenever the thresholds change. If a task's efficiency drops below the new target_eff, its allocation is halved and state changed to DEC; if efficiency exceeds the new value of high_eff, its allocation is doubled and state changed to INC; otherwise its state is maintained as STABLE.

For our experiments, the efficiency thresholds were chosen as follows. If number of tasks is

$$< 4, high\_eff = 0.6, target\_eff = 0.4$$
$$>= 4 and <= 8, high\_eff = 0.8, target\_eff = 0.65$$
$$> 8, high\_eff = 0.9, target\_eff = 0.7$$

These values were selected using the efficiency achieved by each of our benchmarks on each core count, and with the desire to keep the grid of cores as occupied as possible. Finally, the timer interval was chosen to be 100K cycles to strike a balance between letting the tasks reach their stable state as soon as possible, and allowing the scheduler to evaluate the tasks accurately.

5 Experimental Methodology

To explore the benefits of flexibility and to gain insights into the potential for scheduling for an FCMP like TFlex, we compare the performance of some algorithms for TFlex to the best known algorithms for fixed-core CMPs. This section describes the methodology used in our experiments.

Simulator parameters: We model a 32-core TFlex processor. The experiments were done using the cycle accurate simulator also used in [9]. The architectural parameters for a single TFlex core are described in table 1, which was also taken from [9].

Modeling fixed-core CMPs: Fixed-core CMPs are modeled by "freezing" a TFlex configuration. For example, a four-processor SCMP can be created by configuring TFlex to have four processors with eight cores each. We evaluate SCMPs with granularity varying from one 32-core processor to 32 one-core processors. We also evaluate a coarse-grained and a "balanced" ACMP. The coarse-grained version is composed of one P-16, one P-8, and two P-4s, whereas the balanced ACMP is composed of one P-8, two P-4s, four P-2s, and eight P-1s. The latter is called balanced because 16 of the 32 cores are devoted to larger processors, and the remaining 16 are devoted to smaller processors.

Assumptions about reconfiguration: The scheduling algorithms for TFlex described so far only go as far as finding a suitable core count for each task on the sytem. They do not provide any information about *where* to allocate these cores. Here we describe our solution and assumptions about this problem.

For this study we use the following method for reconfiguration. We assume that all tasks are halted and their state pulled out of the grid of cores. Then we use a simple algorithm that sorts the allocations in descending order of size and starts allocating from the lower left corner of the grid, moving upwards and towards the right. This guarantees a legal allocation since all allocations are powers of two. We also assume the following shapes for the various allocations: a P-16 is 4x4, a P-8 is 4x2, a P-4 is 2x2, and a P-2 is 2x1.

We assume that the register state of a task being reconfigured is moved via memory through spills and fills and costs roughly 500 cycles. This is done one task at a time which makes the total reconfiguration cost equal to the sum of the cost of individual tasks. Finally, this cost is added to *all* the tasks on the system including the ones which never participated in the reconfiguration but still had to wait for it to complete.

Our experiments do not count the overheads of the execution of the actual scheduling algorithm that finds the next allocation. The assumption is that the scheduler need not stop any tasks while running its evaluation algorithm. It halts tasks only after it has decided upon an allocation and is ready to reconfigure.

Workload construction: We use a set of hand-tuned benchmarks that were taken from the EEMBC and Versabench suites and are shown in table 1. Table 2 shows

Parameter	Configuration
Instruction Supply	Partitioned 8KB I-cache (1-cycle hit); Local/Gshare Tournament predictor (8K+256 bits, 3 cycle latency) with speculative updates; Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CTB: 16, BTB: 128, Btype: 256.
Execution	Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP).
Data Supply	Partitioned 8KB D-cache (2-cycle hit, 2-way set-associative, 1-read port and 1-write port); 44-entry LSQ bank; 4MB decoupled S-NUCA L2 cache [8] (8-way set-associative, LRU-replacement); L2-hit latency varies from 5 cycles to 27 cycles depending on memory address; average (unloaded) main memory latency is 150 cycles.
Hand-optimized Benchmarks	7 EEMBC benchmarks (a2time, autocore, basefp, bezier, dither, rspeed, tblook), 2 Versabench (802.11b, 8b10b) [11]

Table 1: Single Core TFlex Microarchitecture Parameters

Benchmark	P-1	P-2	P-4	P-8	P-16	P-32
a2time	0.22	0.45	0.63	1.00	0.68	0.31
autocor	0.16	0.29	0.44	0.65	0.88	1.00
basefp	0.12	0.20	0.38	0.68	1.00	0.82
bezier	0.11	0.22	0.45	0.68	0.99	1.00
dither	0.29	0.60	0.91	1.00	0.86	0.95
rspeed	0.17	0.36	0.63	1.00	0.98	0.93
tblook	0.51	0.73	0.80	0.94	0.95	1.00
802.11a	0.24	0.45	0.79	0.93	1.00	0.84
8b10b	0.23	0.44	0.81	0.92	1.00	0.88

Table 2: Cores-to-WS function for our benchmarks

how the performance of these benchmarks varies with number of cores. The length of each benchmark was chosen to be 20M cycles on a P-1 with the desire of striking a balance between simulation time and simulation quality.

We modeled both fixed-sized workloads and dynamic task arrival. The fixed-sized workloads vary from one to 16 tasks each of which is assumed to be available at time zero. The dynamic workloads follow a poisson process arrival model with an arrival rate of two, four, and six tasks per 6M cycles. These arrival rate values were chosen to simulate a diverse range of system loads. A total of 128 tasks arrive in the dynamic workload experiments.

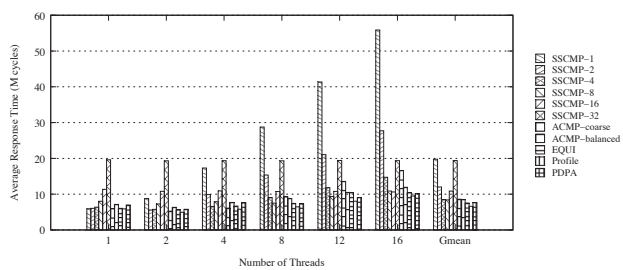
The workloads were generated by randomly selecting benchmarks from our pool of benchmarks. For each size of the fixed workloads, 20 different workloads were generated and the results were averaged.

Metric: We chose *response time* as the metric, which is defined as the time elapsed between a task's departure and arrival. This is particularly relevant for interactive jobs, which care less about their throughput and more about how quickly they can be serviced. It has been one of the metrics of choice in the field of parallel process scheduling [5, 3] and was also used by Kumar et al. to evaluate their scheduling work on ACMPs [10].

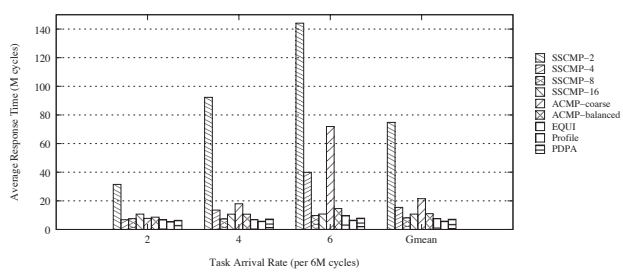
Scheduling algorithms: We evaluate most of the schedulers described in the earlier sections of the paper. For the SCMPs we chose the *FCFS* scheduler that runs the tasks to completion. For the ACMPs we chose *sample-sched*, and for TFlex we evaluate *EQUI, Profile,* and *PDPA*.

6 Experimental Results

This section presents the results of our experiments. We first describe how the results demonstrate the benefits of flexibility by comparing the performance of TFlex and the fixed-core CMPs. This is followed by a comparison of the performance of the scheduling algorithms for FCMPs to help us understand their strengths and weaknesses.



(a) Fixed Workloads



(b) Dynamic Workloads

Figure 4: Performance results

Our experimental results are summarized in Figures 4(a) and 4(b), with average response time on the y-axis as the performance metric (lower is better). Figure 4(a) shows the performance on fixed-size workloads with x-axis representing the workload size (1 to 16 tasks). Figure 4(b) shows the performance on workloads modeling dynamic task arrival with the x-axis representing different task arrival rates (measured in tasks arriving per 6M cycles). The SCMPs

evaluated are labeled as SCMP-n where n is the number of processors in that CMP. For example, SCMP-1 consists of one 32-core processor, SCMP-2 consists of two 16-core processors, etc. The coarse-grained and balanced ACMPs are labeled as ACMP-coarse and ACMP-balanced.

6.1 Benefits of flexibility

Fixed-size workloads: Not surprisingly, Figure 4(a) shows that the best performing SCMP depends directly on the size of the workload. For example, SCMP-2 is the best for size two tasks, SCMP-4 is the best for size 4 tasks, etc. The performance of any given SCMP declines steadily as the size of the workload increases beyond its granularity since tasks are made to wait longer and longer. Moreover, if the size of the workload is much smaller than its granularity, any given SCMP performs worse than those with a more coarse granularity than itself. Overall, SCMP-4 and SCMP-8 perform the best amongst SCMPs, as shown by the geometric mean, labelled *Gmean* in the figure.

Among ACMPs, ACMP-coarse does better for smaller workloads (size one to four) because each processor is more powerful. However, ACMP-balanced is better for larger workloads (eight to 16), since it can accommodate more tasks, significantly reducing the average waiting time.

The benefits of flexibility are evident in the results of the schedulers for FCMPs. There are two components of flexibility–(1) the ability to change the number of processors; (2) the ability to create asymmetrically sized processors. The benefits of the former are shown by how well TFlex performs in comparison to fixed-core CMPs by using just a simple scheduler like EQUI. Its performance is almost as good as the best SCMP for each workload and is on average 10% better overall than the best SCMP (SCMP-8). Not only can this algorithm alter the core allocation depending upon the number of tasks available at time zero, but also as the system load changes due to job completions. EQUI is worse than ACMP-coarse for smaller workloads (smaller than eight tasks) but outperforms it for larger workloads. It outperforms ACMP-balanced for all workload sizes and is about 12% better than the two ACMPs overall.

Profile shows the benefits of creating asymmetry of varying degrees, depending on the number of jobs and the availability of concurrency in each job, and on average outperforms the best SCMP and best ACMP by 21% and 23%, respectively.

Dynamic workloads: Dynamic workloads further demonstrate the benefits of flexible architectures. If the task arrival rate changes such that a large mismatch is created between a fixed-core CMPs granularity and the system load, the performance of fixed CMPs will deteriorate considerably. For example, Figure 4(b) shows that SCMP-4 performs well for a task arrival rate of two (per 6M cycles), but considerably worse for a rate of six, as arriving tasks must wait in the queue because of fewer processors. On the other hand, if the load is much smaller than the granularity, tasks are forced to run on weaker processors despite a lot of the chip resources being idle. The only exception to this is SCMP-16 whose performance does not fluctuate dramatically as the rate changes; however, the performance is still very low for smaller arrival rates since a large number of cores are left idle.

All the FCMP scheduling algorithms are able to adapt to the changes in the system load, since the hardware gives them the option of changing core allocations dynamically. EQUI outperforms the best SCMP (SCMP-8) by 6.3%; the additional capability of creating asymmetry results in Profile outperforming SCMP-8 by almost 32%.

6.2 Comparison of FCMP scheduling algorithms

This section compares Profile, EQUI, and PDPA and provides some insights about their strengths and weaknesses.

Profile: The profiling algorithm performs better than both EQUI and PDPA for all workloads–both fixed and dynamic–because it always executes an optimal allocation assuming no phase behavior in the application. In fact for workloads containing applications with only a single phase, Profile shows us the limits of what a scheduler can achieve. Therefore we compare EQUI and PDPA to this algorithm to gain some insights into their strengths and weaknesses. In workloads that contains programs with phase behavior, Profile would be at a disadvantage, relative to the dynamically adaptive alternatives.

EQUI: As mentioned previously, the performance of EQUI demonstrates that there is merit in adapting to the *number* of tasks even if one does not take into account their individual characteristics. EQUI is trivial to implement and is able to service newly arriving tasks immediately with arrival triggered reconfiguration, thus reducing their waiting time. In fact we observed that it achieves an average waiting time close to zero even for heavy system loads, which for example, can get caused by a task arrival rate of six.

A comparison of EQUI and Profile shows us the additional benefits that the asymmetry component of flexibility can provide. Profile is 14% and 37% better than EQUI for fixed and dynamic workloads, respectively.

The biggest weakness of EQUI is that certain "odd" number of tasks can result in an allocation with too many idle cores. For example, if the system has nine tasks, each will get two cores (since $32/9 = 2$) for a total of 18, resulting in 14 cores being idle. This weakness results in considerable performance loss for this algorithm for the dynamic workload with an arrival rate of six because in this case the system load hovers around eight to 12 tasks. A simple mod-

ification would grow some of the processors to absorb the idle cores and assign the tasks to these asymmetric units, perhaps at random.

PDPA: The dynamic PDPA algorithm has dual strengths of accounting for task characteristics and adapting to system load changes. If a task only has a single phase, it has another advantage of quickly tuning up to its optimal allocation and maintaining it. However, if it does experience phase behavior, PDPA could allocate cores suboptimally since once a task enters the STABLE state it cannot leave until the efficiency thresholds change.

The biggest weakness of this algorithm is that it is a bit conservative. On many occassions cores are left idle because tasks cannot meet the high_eff threshold. For this reason, PDPA performs 15% worse than Profile on fixed workloads and 21% worse on dynamic workloads, overall. A secondary reason for performance loss is that in comparison to Profile and EQUI, PDPA allocates more powerful cores much more slowly. For example in our experiments, at least 300K cycles are required for a task to get eight cores allocated because any allocation changes happen once in 100K cycles, and all tasks get one core initially. This can result in a performance loss of about 3-5%.

The combination of the two weaknesses described above cause EQUI to outperform PDPA for the fixed workloads. However, for the dynamic workloads, PDPA outperforms EQUI since the latter can cause a lot of cores to be idle for certain system loads. This algorithm needs more investigation to find a relationship between efficiency thresholds and load changes that results in more aggressive allocations without being less robust. The challenge here is that on the one hand adjust to fine changes in the load is preferable, but on the other hand there is a risk of doing too many reconfigurations and not finding a stable state.

7 Conclusion

Emerging flexible-core CMP (FCMP) architectures provide new challenges and opportunities for resource management and scheduling. In this paper, we presented three algorithms for scheduling tasks on a FCMP that can adapt to workloads with varying number of tasks and varying degrees of concurrency within the tasks. Our results show that the FCMP and these algorithms can outperform fixed-core CMPs by 21% to 13 times, depending on the workload. A substantial fraction of this improvement stems merely from the capability to match the number of cores to the number of tasks, rather than letting processors or tasks idle in a fixed core CMP. In our experiments, additional but smaller benefit can be achieved by exploiting the asymmetry in the ILP demands of different tasks. Our ultimate goal is to devise systems that obtain the best possible performance, power, and efficiency in general purpose computing by co-optimizing both the hardware and software. While FCMPs and their scheduling algorithms are a step in that direction, we anticipate that combining these techniques with other dynamic resource allocation mechanisms such as DVFS will yield further opportunities for system optimization.

References

[1] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *International Symposium on Computer Architecture*, pages 506–517, June 2005.

[2] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7), 2004.

[3] J. Corbalan, X. Martorell, and J. Labarta. Performance-driven processor allocation. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):599–611, 2005.

[4] D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling – a status report. In *Workshop on Job Scheduling Strategies for Parallel Processing*, June 2004.

[5] D. G. Feitelson. Job scheduling in multiprogrammed parallel systems. Technical Report RC 19790 (87657), IBM Research, August 1997.

[6] T. Ibaraki and N. Katoh. *Resource allocation problems: Algorithmic approaches*. MIT Press, 1988.

[7] E. Ipek, M. Kirman, N. Kirman, and J. F. Martínez. Core fusion: accommodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture*, pages 186–197, June 2007.

[8] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[9] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, S. W. Keckler, and D. Burger. Composable lightweight processors. In *International Symposium on Microarchitecture*, December 2007.

[10] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *International Symposium on Computer Architecture*, pages 64–75, June 2004.

[11] R. M. Rabbah, I. Bratt, K. Asanovic, and A.Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. Massachusetts Institute of Technology Technical Report MIT-LCS-TM-646, 2004.

[12] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, 2003.

[13] D. Tarjan, M. Boyer, and K. Skadron. Federation: Out-of-order execution using simple in-order cores. Technical Report CS-2007-11, University of Virginia, Department of Computer Science, August 2007.

[14] D. W. Wall. Limits of instruction-level parallelism. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, 1991.

[15] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *International Symposium on High Performance Computer Architecture*, pages 25–36, February 2007.