Multitasking Workload Scheduling on Flexible-Core Chip Multiprocessors

Divya P. Gulati
University of Texas at Austin
1 University Station C0500
Austin, Texas 78712
dpgulati@cs.utexas.edu

Changkyu Kim
Intel Corporation
3600 Juliette Ln, SC12-303
Santa Clara, California 95054
changkyu.kim@intel.com

Simha Sethumadhavan
Columbia University
1214 Amsterdam Ave.
New York, New York 10027
simha@cs.columbia.edu

Stephen W. Keckler
University of Texas at Austin
1 University Station C0500
Austin, Texas 78712
skeckler@cs.utexas.edu

Doug Burger
University of Texas at Austin
1 University Station C0500
Austin, Texas 78712
dburger@cs.utexas.edu

ABSTRACT

While technology trends have ushered in the age of chip multiprocessors (CMP), a fundamental question is what size to make each core. Most current commercial designs are symmetric CMPs (SCMP) in which each core is identical and range from a simple RISC processor to a complex out-of-order x86 processor. Some researchers have proposed asymmetric CMPs (ACMP) consisting of multiple types of cores. While less of an issue for ACMPs, the fixed nature of both these architectures makes them vulnerable to mismatches between the granularity of the cores and the parallelism in the workload, which can cause inefficient execution. To remedy this weakness, recent research has proposed flexible-core CMPs (FCMP), which have the capability of aggregating multiple small processing cores to form larger logical processors. FCMPs 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 problem, describes the challenges associated with it, and evaluates algorithms appropriate for multitasking on FCMPs. We also evaluate static-core CMPs of various configurations and compare them to FCMPs for various multitasking workloads.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*multiprocessing/multiprogramming/multitasking, scheduling*; C.1.2 [**Processor Architectures**]: Multiprocessors—*parallel processors*

General Terms

Algorithms, Experimentation, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.
Copyright 2008 ACM 978-1-60558-282-5/08/10 ...$5.00.

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 demonstrate substantial diversity. For example, the amount of ILP available across different applications may vary widely. Even the characteristics of a single program may vary during different phases of its execution [19]. Selecting the number of cores and their size 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 than SCMPs. However, the composition of the ACMPs must still be determined at design time, leaving them vulnerable to mismatches between the workload and the system.

Recently proposed alternatives to static-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 [12, 14, 20]. 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 [12], and better tolerance to performance asymmetries resulting from the fixed but varying

cores [2]. 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 quite large.

While providing flexibility is one challenge for FCMPs, it is also challenging to manage their resources. A scheduler and resource allocator must determine (1) how many logical processors to assemble from the cores, (2) how large each processor should be, (3) what topology each logical core should take, (4) where each task should run, (5) under what circumstances should the configuration or 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 scheduling algorithms. We envision FCMPs to be used in a variety of ways including for multiprogrammed multi- and single-threaded workloads. As a first step, we explore the problem in an environment consisting of multiprogrammed single-threaded workloads with both fixed and dynamic workloads, and leave the exploration of multi-threaded workloads for future work. We adapt scheduling algorithms from the multiprocessor scheduling literature to FCMPs and compare them to existing algorithms for symmetric and asymmetric CMPs. The purpose of this comparison is to determine if FCMPs outperform static-core CMPs given real scheduling algorithms and workloads, and if so, by how much. In our experiments, we focus solely on performance, for which we use *makespan*, defined as the total execution time of a workload, and *response time*, defined as the time elapsed between a task's arrival and departure, as the metrics. Our results show that a FCMP's ability to adapt to task count provides a benefit of 14% and 23% over the best static-core CMP for fixed and dynamic workloads, respectively; the ability to adapt to task types increases the benefits by 13% and 42%, respectively. For simplicity, we restrict this study to composition of the cores and leave an examination of DVFS for future work.

2. STATIC-CORE ARCHITECTURES

With conventional uniprocessor architecture scaling coming to a close, microprocessor researchers and vendors have turned to multicore architectures. Depending on the target workload, different vendors have chosen to optimize for different core sizes, which we term *bulldozers*, *chainsaws*, and *termites*. In the general purpose space, vendors are building bulldozers, such as the 4-issue out-of-order AMD quad-processor Barcelona, which uses about $36mm^2$ per core in a 65nm technology [5]. In the commercial space where parallel transaction processing is critical, vendors are building chainsaws, such as the Sun's single-issue in-order Niagara-2, which uses 14 mm^2 [17]. Intel's Polaris is an example of a collection of termites in which each processor is little more than a floating-point unit and a router, occupying $2.5mm^2$ [21]. The granularity and number of the cores affects the type of concurrency exploitable by the processor: bulldozers are better for serial or coarse-grained parallel workloads, while termites are better for fine-grained parallel workloads. The goal of flexible-core architectures is to build termites or chainsaws, and aggregate them together to create bulldozers when necessary.

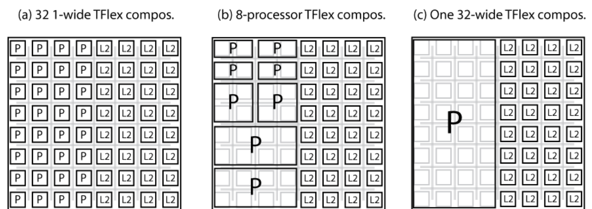


Figure 1: Three CLP compositions.

Recently, architecture researchers have proposed heterogeneous multicore (also known as asymmetric multicore) processors as a means to improve power efficiency or performance over symmetric multicore architectures. Kumar, et al. initially proposed a single-ISA heterogeneous multicore architecture, which consisted of instances of different generations of Compaq Alpha processors [15]. They suggest that energy efficiency can be maximized by running a program on the least-powerful core required to meet its performance demand. Others, such as Ghiasi and Grunwald [9] and Grochowsky, et al. [10] also exploit heterogeneity in multicore architectures for power efficiency. Kumar, et al. extended their work to the multiprogramming space, determining that the core asymmetry provided performance benefits as well [16]. Annavaram et al. examine the viability of heterogeneity to accelerate multithreaded programs [1]. Heterogeneity is also finding its way into the commercial sphere. For example, IBM's cell processor includes one PowerPC processor for executing system code and 8 SPUs for executing threaded and data-parallel code [18]. We expect this trend to continue as specialized cores for graphics or cryptography are incorporated into processor dies. While efficient for their assigned tasks, such specialized processors do not provide adaptivity to different granularities of parallelism as flexible-core processors can.

3. FLEXIBLE-CORE ARCHITECTURES

Flexible-core architectures aim to provide adaptivity in the number and granularity of processors, enabling the system to efficiently execute both a large and a small number of tasks. 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 [12]. Federation is a similar solution, but instead federates multiple in-order cores to create an out-of-order processor [20]. 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 [22]. 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. The algorithms explored in this paper are, in principle, applicable to the aforementioned FCMPs. However, these FCMPs have limited scalability (2-4 cores). TFlex

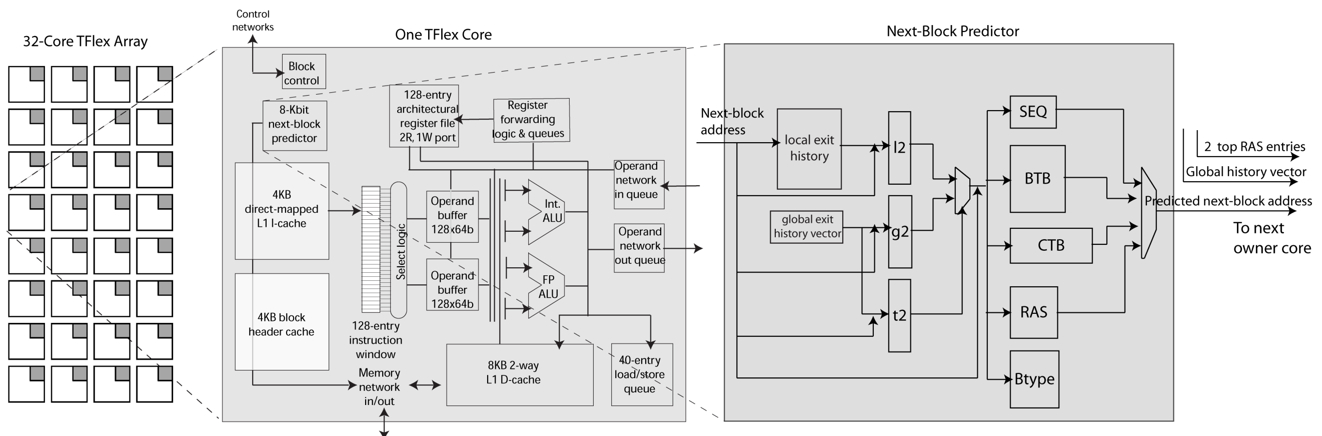


Figure 2: Illustration of microarchitectural components of one core of a 32-core TFlex CLP (left and center) and internal organization of the next-block predictor (right).

processors can be composed of more processors and thus will benefit from more sophisticated scheduling algorithms.

In this paper, we use the TFlex Composable Lightweight Processor (CLP) architecture [14]. Each CLP core is a simple dual-issue out-of-order processor. TFlex also contains 32 banks of L2 cache arranged in a non-uniform (NUCA) architecture [13]. Figure 1 (taken from [14]) shows a high-level floorplan of this processor, with three of many possible compositions: (a) 32 1-core processors, (c) one 32-core processor, and (b) a mix of processors composed of different numbers of cores. Throughout this paper we will use terminology of P-N to refer to a logical processor consisting of N cores. The rest of this section describes the architecture of TFlex, how cores are composed into larger processors, and considerations for recompositions.

ISA support: The scalability of FCMP architectures can be hindered by execution models that requires a sequential instruction stream fetched from a common instruction store. The model also influences the ease of distributing microarchitecture structures, such as the operand bypass bus, 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 [3].

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, which are 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 inherent in a 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 in a specific order across all cores composed as one processor, which helps to locate them using a point-to-point network. The interleaving changes when the core allocation changes.

Microarchitectural support: TFlex achieves full composability, which means that no structures are shared across cores. Figure 2 (from [14]) 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. *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. *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. *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. Adjustments to the hashing functions could allow for non-power-of-2, but we have not experimented with such compositions. 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 TFlex. Additional details of the TFlex architecture are found in [14].

Processor recomposition: TFlex contains a virtualization layer that allows creation of arbitrary compositions. Each core is assigned a *physical ID*, while each core within each logical processor is assigned a *logical ID*. Each core contains a composition map that translates its logical ID to physical ID so that the control networks implement the processor control protocols correctly. The processor size is used to compute interleaving factors. Recomposition of a processor requires three steps: stopping the pipeline, moving register state from the old set of cores to the new set of cores according to the new interleaving, and adjusting the composition maps to reflect the new mapping of physical cores to 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. TFlex provides cache coherence at the L1/L2 interface, which makes invalidating the L1 D-cache unnecessary.

4. SCHEDULING STATIC-CORE CMPS

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 can most trivially be implemented in a first-come, first-served (FCFS) algorithm, with periodic rescheduling to ensure fairness. 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 [7]. For symmetric CMPs, we assume that tasks are independent of one another and use a simple FCFS 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.

Asymmetric CMPs: Scheduling is more challenging when the cores in a CMP have different characteristics as the scheduler must not only decide which tasks to run, but also on which core to run them. While ACMPs are relatively new, Kumar et al. have examined a family of scheduling algorithms for them, which can be placed into three categories: static, random, and dynamic [16]. The static algorithm uses a priori profiling information about each task to find an optimal assignment of tasks to processors; the assignment is updated on task arrival/departure. The random algorithm simply finds a random assignment but ensures that more powerful cores get used before less powerful ones. The dynamic algorithms adapt to changes in the environment that cannot be predicted a priori (like task phase changes). Kumar, et al. divide their dynamic algorithm into two phases: *sample* and *steady*. In the sampling phase the scheduler tries different assignments to find the "best" one, which is then run in the much longer steady phase.

Sampling algorithms: Kumar et al. describe several sampling algorithms, but we describe here only their best performing one called *sample-sched*. Given n tasks, sample-sched runs at most $4 * n$ different assignments of tasks to processors, which are chosen randomly but ensure that each task runs at least once on the weakest processor. The assignment with the best *weighted speedup* (WS), which is the sum of individual speedups of each task for a given configuration, is selected to run in the next steady phase. Individual speedups use the performance on the weakest processor as the baseline.

Triggering sampling: Kumar et al.'s best policy, *bounded-global-trigger*, triggers sampling if the sum of the percentage IPC change of all tasks running exceeds 100%. To guard against short phase behavior, the algorithm delays sampling until the 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 sampling. We augment this algorithm by sampling if a task arrives/departs and the lower threshold has been exceeded and allocating idle cores to a waiting task if arrival/departure occurs in the steady phase.

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. These parameters are relatively small in order to accommodate the slow execution rate of simulation. While these values are not as large as those chosen by Kumar et al., their ratios are similar to those in the original study.

5. SCHEDULING 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. One configuration might include four logical processors: one P-16, one P-8, and two P-4s. The number and size of the processors is influenced by the number of tasks on the system, the parallelism profiles of the tasks, and by the degree of contention for shared resources like the L2 cache.

(2) Determining the topology of each logical processor: FCMPs 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: The objective is to find a free location for each task without shuffling already running tasks. Fragmentation makes this challenging because tasks that terminate leave vacated cores that can occupy discontiguous regions of the grid. Assigning cores contiguously to a task would require shuffling other tasks around. The problem of finding logical processors has been studied for multiprocessors, an excellent survey of which is provided by Feitelson in [7].

(4) When should the configuration change: The scheduler attempts to minimize time spent in a non-ideal configuration. 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. A tradeoff exists between running in a non-ideal configuration and the overhead of repeated 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 one. As described in Section 3, 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, which is similar to a context switch, is to spill the registers to memory, reconfigure the processors, and retrieve the register values from memory. The second method minimizes the memory traffic by transmitting the register values directly from the old mappings to the new ones. This 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 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 selecting the right size and number of

the logical processors. To this end, we examine configuration scenarios that do not encounter the issues of fragmentation and shape. We classify the space of FCMP scheduling algorithms into two categories: (1) adapting to changes in task count; (2) adapting to differences in task type.

5.1 Algorithms adapting to task count

We define this class of algorithms as those that ignore the characteristics of individual tasks, but do account for the number of running tasks. One simple algorithm divides the number of cores into logical processors of equal size and assigns each one to a task. 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 available cores and can change the task allocation. For example in a 32-core system with two tasks, each will get sixteen cores. We call this algorithm **FSCMP**, which stands for Flexible Symmetric Chip Multiprocessor.

FSCMP has the drawback of leaving cores idle if the total core count is not evenly divisible by the task count. For example, in a 32-core system, a task count of nine will cause each task to get $32/9 = 2$ cores (assuming power-of-two sizes), thus resulting in 14 idle cores. A simple optimization is to divide the idle cores equally among a subset of the tasks. So in the previous example, this optimization will result in seven tasks getting four cores each, and two tasks getting two cores each. We call this algorithm **FACMP-Equi**, where FACMP stands for Flexible Asymmetric CMP since this algorithm may create asymmetric allocations. A slightly different version of this algorithm has been used in a number of studies on scheduling for parallel systems [7].

5.2 Algorithms adapting to task type

This class of algorithms not only adapts to changes in task count, but also accounts for the characteristics of tasks. Information about the characteristics of tasks can be collected either off-line through profiling, or dynamically on-line. We describe one algorithm of each type below. We term these algorithms as FACMPs as well because of their ability to construct asymmetric allocations.

Profile-based algorithms: This class of algorithms assumes that some information about the characteristics of each task is available to the scheduler at arrival time. While this information could be obtained through a priori or on-line profiling, 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 [6], we have not yet explored this algorithm.

We call the profile-based algorithm that we implement and examine in this paper **FACMP-Profile**. This algorithm assumes that information about how a task's weighted speedup varies with core count (cores-to-speedup function) is available at task arrival time. Given this information, scheduling

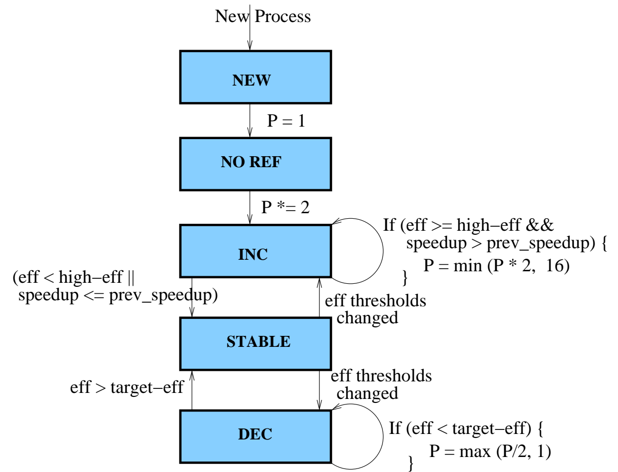


Figure 3: FACMP-PDPA state diagram.

can be stated as the following optimization problem:

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

where f_i is the cores-to-speedup 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 [11]. 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 FACMP-Profile exploits knowledge of the characteristics of an individual task, it does not account for any phase behavior exhibited by a task. Furthermore, 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.

The dynamic algorithm **FACMP-PDPA** (Performance Driven Processor Allocation) is adapted from [4]. Our modified version of the algorithm allocates cores based on how efficiently a task is executing. 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 a predefined threshold called *target_eff*, cores are taken away. Otherwise the allocation is maintained. The efficiency thresholds are chosen such that target_eff < high_eff. The algorithm can be best understood by following the state diagram shown in Figure 3. Each task exists in some state of this state machine. A timer interrupt periodically triggers invocation of the scheduler, which evaluates each task on the system as follows.

- **NEW:** Arriving tasks are placed in a wait queue.
- **NOREF:** A task selected to execute is allocated one core providing the baseline for later calculating a task's efficiency, which is defined as (Speedup_over_1_core) / (Number_of_cores_currently_allocated). Once done, the task moves to INC with its core count doubled.
- **INC:** If a task's efficiency and speedup exceed high_eff and previous speedup, respectively, its allocation is doubled and it stays in INC. Otherwise it is moved to STABLE and if its efficiency drops below target_eff, its allocation is halved. Comparison of current and

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 [13] (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 100 cycles. 2 SDRAM controllers each with peak bandwidth of 8-bytes per processor cycle.
SPEC2000 Benchmarks	High-ILP: swim, mgrid, gcc, apsi, art, sixtrack; Low-ILP: applu, bzip2, parser, ammp, twolf, mcf

Table 1: Single core TFlex microarchitecture parameters.

previous speedup ensures that we do not allocate more cores if speedup fell despite a high efficiency;

- **DEC:** If eff < target_eff, core count is halved and the task stays in DEC; else it moves to STABLE.
- **STABLE:** Once a task enters this state, it stays unless the efficiency thresholds change. The thresholds change if the system load changes significantly, which ensures that a task does not oscillate between states. If efficiency < target_eff, the allocation is halved and the task is moved to DEC; if efficiency > high_eff, the allocation is doubled and the task moves to INC.

New tasks are always serviced first to ensure that they at least get started; remaining tasks are then serviced in FCFS order. If an increment in a task's allocation cannot be satisfied, the scheduler ignores it and moves to the next task. The allocation is always between one and 16 cores. The algorithm tries to allocate as many cores as close to ideal (IPC maximizing) for a task, while taking into account the load on the system.

The efficiency thresholds were chosen as follows:
If num tasks < 3, high_eff = 0.41, target_eff = 0.23,
If num tasks < 7, high_eff = 0.65, target_eff = 0.41,
If num tasks < 10, high_eff = 0.77, target_eff = 0.65,
If num tasks < 13, high_eff = 0.89, target_eff = 0.77,
Otherwise high_eff = 0.99, target_eff = 0.89

These values were selected using the efficiency achieved by our benchmarks on each core count, and with the desire to keep the grid as occupied as possible. For example, if num-tasks < 7 but ≥ 3, a P-8 becomes a luxury and only tasks with a relatively high efficiency for a P-4 (the next smaller processor) should get them (high_eff = 0.65). To ensure that they keep it, we set the target_eff to the efficiency these tasks achieve on a P-8 (target_eff = 0.41). As load increases, the thresholds increase to allow only the highest efficiency tasks get more cores. 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.

6. EXPERIMENTAL METHODOLOGY

We model a 32-core TFlex processor using the cycle-accurate simulator used in [14]. Table 1 shows the architectural parameters of a single TFlex core.

Modeling static-core CMPs: Static-Core CMPs are modeled by "freezing" a TFlex configuration. For example, a four-processor static-symmetric CMP (SSCMP) can be created by configuring TFlex to have four processors with eight cores each. We evaluate SSCMPs with granularity varying from two P-16 processors to 16 P-2 processors. We also evaluate coarse-grained, fine-grained, and "balanced" static-asymmetric CMPs (SACMP), where balance indicates that half of the processors are large and half are small. Table 2(a) lists all the architectures/algorithms we modeled.

Reconfiguration: All tasks are halted during reconfiguration and their state pulled out of the array of cores. Allocations are sorted in descending order of size and placed starting from the lower left corner of the array, moving upwards and to the right. This guarantees a legal allocation since all are powers-of-two. The following shapes are assumed: a P-16 is 4x4, a P-8 is 4x2, a P-4 is 2x2, and a P-2 is 2x1. Tasks whose register state is pulled out or pushed in to the array incur a *read* and *write* overhead, respectively. We estimate each to be 150 cycles since the cost of writing the 128 registers to L2 can be amortized. To account for any L2 misses generated while accessing the register state we add a margin of error of 100 cycles. Updating the configuration map costs a further 50 cycles. The total adds up to roughly 500 cycles of overhead. We assume that tasks are re-configured sequentially, so each task involved in the reconfiguration is charged the sum of the overheads of all such tasks. Note that we do simulate the loss of L1 locality and its impact on performance due to reconfiguration. We assume that the scheduling algorithm runs in parallel with the tasks, thus avoiding any additional overhead.

Workload construction: We use a subset of the SPEC CPU2000 benchmarks (Table 1), selected with the desire to have diversity in ILP and the memory footprint. Table 2(b) shows how their performance varies with number of cores. For each benchmark, the number of blocks was chosen such that it ran for 20M cycles on a single core, in order to strike a balance between simulation time and fidelity. We modeled both fixed-sized workloads and dynamic process arrival. The former vary from two to 16 tasks, each of which are assumed to be available at time zero. For each size, ten different workloads were generated and the results averaged (geometric mean). The dynamic workloads follow a Poisson process arrival model, with a rate of two, four, and six tasks arriving in 6M cycles, and a total of 128 arrivals. These values were chosen to simulate a variety of system loads. All algorithms handle the case where number of tasks exceeds the number of cores in the system. The workloads were constructed by random selections from our pool of benchmarks.

Metrics: (1) *Makespan:* We chose makespan, which is defined as the amount of time to complete all tasks in a workload, as the metric for fixed workloads. For a workload of a given size this is equivalent to throughput [8]. (2) *Average Response Time:* We chose response time as the metric for dynamic workloads, which is defined as the time elapsed between a task's departure and arrival, and includes any waiting time. This metric is particularly relevant for interactive jobs, which care less about their throughput or execution time, but more about how quickly they can be ser-

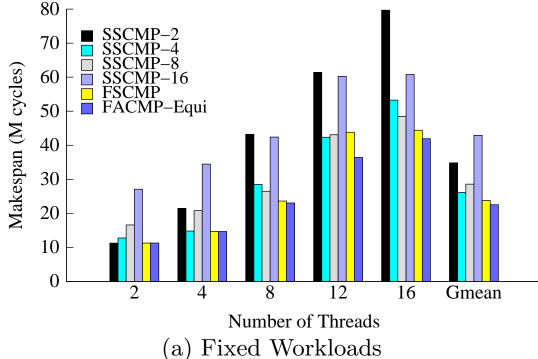
(a)

Label	Description
SSCMP-2	Static Symmetric CMP: two P-16s
SSCMP-4	Static Symmetric CMP: four P-8s
SSCMP-8	Static Symmetric CMP: eight P-4s
SSCMP-16	Static Symmetric CMP: sixteen P-2s
SACMP-Coarse	Static Asymmetric CMP: one P-16, one P-8, two P-4s
SACMP-Balanced	Static Asymmetric CMP: two P-8s, two P-4s, four P-2s
SACMP-Fine	Static Asymmetric CMP: four P-4s, four P-2s, eight P-1s
FSCMP	Flexible Symmetric CMP: uses equal core allocation
FACMP-Equi	Flexible Asymmetric CMP: uses equal allocation w/out idle cores
FACMP-Profile	Flexible Asymmetric CMP: uses off-line profiles of tasks
FACMP-PDPA	Flexible Asymmetric CMP: uses efficiency thresholds

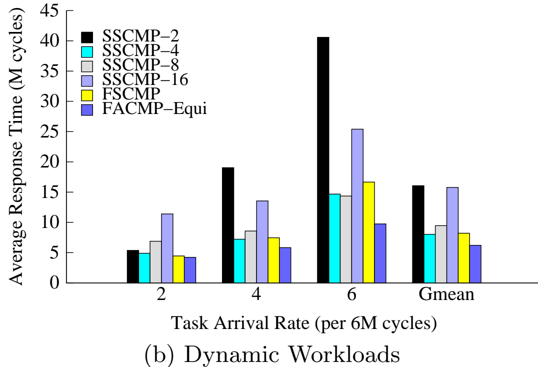
(b)

Benchmark	P-1	P-2	P-4	P-8	P-16	P-32
swim	0.60	1.18	2.00	3.85	7.01	10.86
mgrid	0.69	1.21	2.08	3.05	5.02	5.91
gcc	0.82	1.66	2.56	3.30	3.59	4.32
apsi	0.88	1.82	2.76	4.00	4.18	3.62
art	0.74	1.26	2.29	3.26	3.72	3.95
sixtrack	0.76	1.52	2.38	2.98	3.29	2.23
applu	0.69	1.36	2.07	2.57	2.66	1.71
bzip2	0.85	1.59	1.78	1.68	1.47	1.49
parser	0.60	0.97	1.27	1.62	1.75	1.58
ammp	0.78	1.26	1.57	1.60	1.30	0.98
twolf	0.46	0.84	1.14	1.27	1.29	1.20
mcf	0.22	0.28	0.31	0.39	0.40	0.38

Table 2: (a) Summary of architectures and algorithms modeled; (b) Cores-to-IPC for the benchmarks.



(a) Fixed Workloads



(b) Dynamic Workloads

Figure 4: Adapting to change in task count.

viced. Moreover, for dynamically arriving tasks *throughput* cannot exceed the rate of arrival, which reduces its value as a metric for such workloads. Response time and throughput have been the metrics of choice in the field of parallel systems scheduling [4, 7], and were also used by Kumar et al. to evaluate their scheduling work on ACMPs [16].

7. EXPERIMENTAL RESULTS

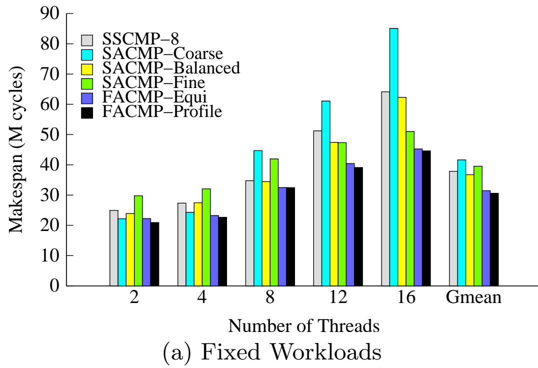
This section present the benefits of adapting to changes in task count and to adapting to different task types. Then, we compare different scheduling algorithms for FCMP architectures and explore the sensitivity of the FCMP scheduling algorithms to L2 cache interference.

Adapting to changes in task count: To focus on the benefits of adapting to task count changes, we compose workloads of only high-ILP benchmarks, which reduces inter-task diversity. Moreover, we limit the experiment to symmetric architectures by comparing SSCMPs to FSCMP and FACMP-Equi. We include FACMP-Equi to compensate for the fact that FSCMP does not always match the task count. While FACMP-Equi generates some asymmetry, the algorithm does not make intelligent use of it.

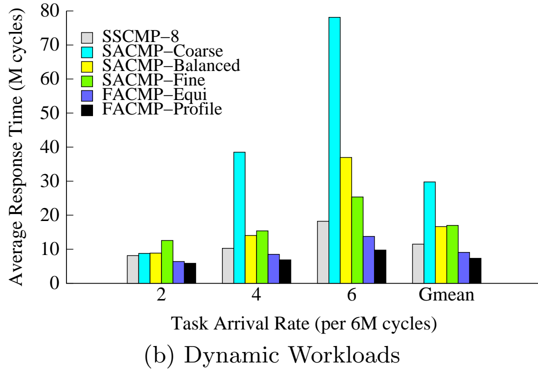
Figure 4(a) shows the benefits for fixed workloads. The x-axis is the workload size (tasks) and the y-axis is the metric *Makespan*; lower bars indicate better performance. For each workload size, the first four bars represent the SSCMPs we evaluated, and the next two correspond to FSCMP and FACMP-Equi. The SSCMPs that match the task count precisely generally perform the best for that task count. The flexible-core algorithms outperform even the best SSCMPs since they adapt to the task count as it changes due to task completions. Not matching the processor and task counts *precisely* causes FSCMP to perform about 5% worse than FACMP-Equi, and worse than SSCMP-4 and -8 for workloads of size 12. Overall, FACMP-Equi outperforms SSCMPs by 14–47%. SSCMP-16 performs worse than both SSCMP-4 and -8 for workloads of size 16 despite matching the task count precisely because (1) the higher degree of concurrency in SSCMP-16 exacerbates contention in the L2 cache, and (2) smaller L1 caches cause more misses. The latter is less of an issue for FSCMP which allocates more cores to the tasks still running as tasks finish. This policy eliminates about 5% of L2 accesses and 7% of L2 misses.

Figure 4(b) shows the benefits of adapting to task count for the dynamic workloads–the x-axis is the task arrival rate per 6M cycles, and the y-axis is the metric *Average Response Time*. Lower bars indicate better performance. FACMP-Equi outperforms all SSCMPs by 23–61%. The inability of FSCMP to precisely match task count is exacerbated in the dynamic workloads since there are more opportunities for this mismatch to occur; thus FSCMP performs worse than SSCMP-4 overall. Interestingly SSCMP-16 is still worse than SSCMP-4 for an arrival rate of six. For our benchmarks SSCMP-4 has a 3.25 times faster average execution time than SSCMP-16, but since SSCMP-16 has four times more parallelism, the time in which SSCMP-16 completes 16 tasks, SSCMP-4 should complete $(3.25/4) * 16 = 13$ tasks. The proximity of the two numbers suggests that the average waiting times of the two SSCMPs should be close, which is what happens in reality with SSCMP-4 and SSCMP-16 having an average waiting time of 12.25M cycles and 10M cycles, respectively. However, SSCMP-4's much faster execution time dominates the response time and causes SSCMP-4 to outperform SSCMP-16.

Adapting to differences in task type: To have diver-

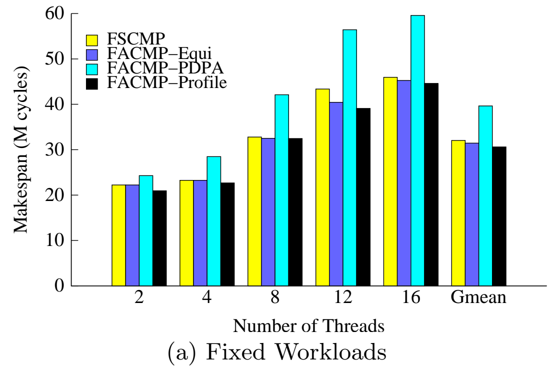


(a) Fixed Workloads

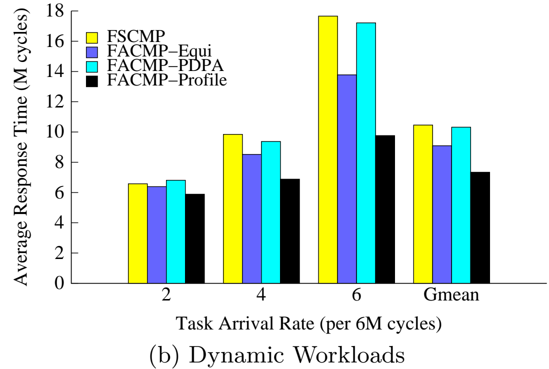


(b) Dynamic Workloads

Figure 5: Adapting to task type.



(a) Fixed Workloads



(b) Dynamic Workloads

Figure 6: Comparison of flexible-core algorithms.

sity across tasks, the workloads selected for this comparison have an equal mix of high- and low-ILP benchmarks. Figure 5 shows results of the comparison between CMPs with FACMP-Equi and -Profile. FACMP-Equi highlights the advantages of adapting to task count; FACMP-Profile shows the additional benefits provided by the ability to adapt to task type. We include all three SACMP variants and SSCMP-8, which is the best SSCMP across all workloads.

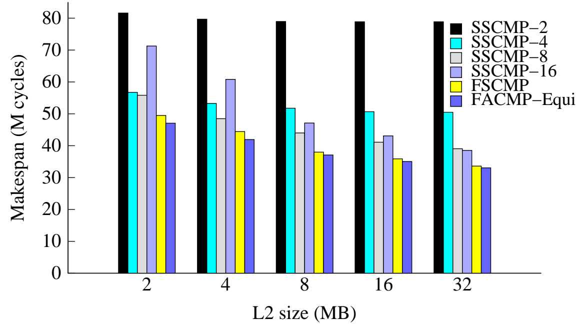
For the fixed workloads, the ability to adapt to task type provides a benefit of 13%, 8%, 14%, and 9% over SSCMP-8, SACMP-Coarse, SACMP-Balanced, and SACMP-Fine, respectively. This benefit is larger for the dynamic workloads: 42%, 8%, 19%, and 18%, respectively, mainly due to the difference in the metrics–*response time* captures the benefit experienced by *each* task, while *makespan* hides it since it is determined solely by the task finishing last.

Surprisingly SSCMP-8 outperforms all SACMPs for the dynamic workloads. As an example, SACMP-Balanced has half the performance of SSCMP-8 for an arrival rate of six. Breaking response time into execution and waiting time, we find that the SACMP's waiting time is thrice that of the SSCMP's. This difference is a result of the repeated shuffling of tasks in the sampling phase of the SACMP algorithm, which fragments the execution of tasks, and may cause them to finish later than they otherwise would. This effect can be understood by the following example. Assume that two identical tasks with unit execution time need to run on a uniprocessor. One schedule runs them sequentially, which results in the first task completing at time 1 and the second at time 2. Another schedule runs each for 1/2 time units before swapping with the other. This schedule results in the two tasks completing at time 1.5 and 2 units, respectively. Although the total execution time for both schedules is the same, the second has an inferior average response time. Other reasons for the worse performance of SACMPs are: (1) each sampling phase incurs the overhead of multiple reconfigurations, and (2) sampling increases the number of tasks sharing the L2, thus exacerbating contention.

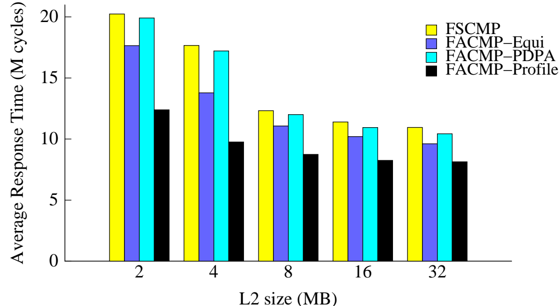
Comparison of flexible-core scheduling algorithms: The only difference between FSCMP and FACMP-Equi is that the former can sometimes create more processors than necessary, which results in idle cores. Between Profile and Equi, the former primarily outperforms the latter due to its ability to adapt to task type. Figure 6 shows that the overall benefits of adapting to task type for fixed and dynamic workloads are about 3% and 19%, respectively. A higher value for dynamic workloads can again be attributed to the difference in the metrics of the two classes of workloads. For the dynamic workloads the relative performance of Profile over Equi improves as the task arrival rate increases, because Profile gives priority to high-ILP tasks, causing fewer tasks to execute concurrently than in Equi. This strategy reduces L2 accesses and L2 contention by allocating larger L1 caches to each task. The percentage of L2 misses saved by FACMP-Profile over FACMP-Equi increases from 2% for arrival rate of two to 22% for arrival rate of six. This improvement in cache performance causes a reduction in execution time that outweighs the additional waiting time.

Since PDPA has both attributes of flexibility without relying upon the availability of a priori information about tasks, we expected that it would bridge the gap between Equi and Profile. Surprisingly, PDPA performs worse than Equi for both fixed and dynamic workloads, which can be attributed to the following two disadvantages.

First, allocating cores based on global efficiency thresholds results in inappropriate allocations for many tasks. During



(a) Fixed Workloads



(b) Dynamic Workloads

Figure 7: L2 cache sensitivity.

low system load, the objective is to allocate a task its ideal number of cores, while for high system load the goal is to utilize all cores without causing disproportionate allocations between tasks. These two objectives are virtually impossible to achieve by using global efficiency thresholds.

Second, the desire to quickly discover and maintain a stable state for each task has some drawbacks. When the system is in the stable state, it cannot immediately absorb any cores that become free when tasks terminate. The only way to leave the stable state is for the efficiency thresholds to change, which happens only if the system load changes sufficiently. Moreover, the prevalence of even small phases can cause a task to get "stuck" in the wrong state. The effects are especially drastic for high system loads where some tasks perform very poorly for some period of time due to insufficient L2 capacity and bandwidth, which causes their efficiency value to be much lower than expected. These tasks then experience a form of double jeopardy since the only way for these tasks to become "unstuck" is for the system load to reduce, which cannot happen for a very long time since most of them are stuck with low allocations. Refinements to this class of on-line algorithms will be required to make them feasible for FCMPs.

L2 size sensitivity: Figure 7 shows the result of one sensitivity study: measuring the relative effectiveness of different algorithms as L2 capacity changes. We limit the study to the heaviest workloads which is size 16 for the fixed workloads and arrival rate six for the dynamic workloads. Departing from the earlier fixed 4MB L2 cache, we varied the L2 cache size from 2MB to 32MB without changing its organization. To minimize clutter we focus only on the most relevant algorithms/architectures for each class of workloads. As expected, algorithms providing higher task concurrency are more sensitive to L2 size since they allocate smaller processors, including smaller L1 caches. In the fixed workloads of Figure 7(a), all architectures benefit from an increasing cache size, and FCMPs still outperform the SSCMPs for each cache size. Although SSCMP-16 showed lower performance than both SSCMP-4 and -8 for a 4MB L2 due to excessive L2 contention, it catches up with both when provided with a larger L2. SSCMP-16 catches SSCMP-8 later than -4 because the former achieves a better balance between concurrent task execution and L2 contention. In the dynamic workloads of Figure 7(b), all algorithms benefit from a bigger L2, but due to the inherent differences between them, their relative rank remains the same. Although its performance improves with L2 size, FACMP-PDPA still performs relatively poorly due to the weaknesses described earlier.

Application of FCMP algorithms to static-core CMPs and vice-versa: Comparing the performance of static-core CMPs to that of flexible-core CMPs is complicated by the existence of two sources of differences: architecture and algorithms. While an experimental evaluation that separates these factors is beyond the scope of this paper, we present below some qualitative insights into the viability of the algorithms on different architectures.

The default *sampling algorithm* as implemented in this paper is not likely to perform well on a manycore TFlex system because of the sheer number of possible configurations. To limit the search space, a modified algorithm could bias the types of considered configurations depending upon the system load. For example, the scheduler could consider configurations with coarse-granularity processors at low system load and fine-granularity processors at high system load.

FACMP-PDPA on SACMPs reduces to sorting tasks into a descending order of efficiency and allocating them from coarser- to finer-granularity processors, regardless of the system load. Unlike TFlex, there is no possibility of making fine adjustments to the allocations by changing the granularity of the available processors.

FACMP-Profile can be easily applied to SACMPs but is likely to be less effective than on TFlex since it will only have a single configuration to work with. However, the constraint of a single configuration will make the dynamic programming problem much easier on SACMPs.

8. CONCLUSION

Emerging flexible-core CMP architectures provide new challenges and opportunities for resource allocation and scheduling. In this paper, we examined two benefits provided by FCMPs: (1) the ability to adapt the number of processors to the number of tasks, and (2) the ability to adapt each processor's ILP granularity to tasks with different concurrency requirements. Our results show that moderate core count symmetric CMPs (4 or 8 large processors) work well, but that the flexibility of FCMPs to better match the core count to task count results in about a 20% improvement in multitasking workload latency or response time. When accounting for variation in task type, FCMP's ability to adjust the size and number of cores provides about a 20% boost over the best asymmetric CMP we examined. Finally, simple FCMP scheduling algorithms that attempt to evenly distribute the cores to the tasks work reasonably well, but profiling information about task requirements can improve performance. Our on-line performance-driven processor allocation algorithm performs rather poorly, indicating that challenges remain for effective on-line scheduling.

Our 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. Furthermore, we expect to expand this study to include multi-threaded applications, which not only pose the challenge of determining the allocation to a process but also the sub-allocation to each of its threads. Moreover, thread synchronization issues increase the importance of finding an appropriate physical location for each logical processor.

9. ACKNOWLEDGEMENTS

This research is supported by a Defense Advanced Research Projects Agency contract F33615-01-C-4106 and by NSF CISE Research Infrastructure grant EIA-0303609.

10. REFERENCES

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law Through EPI Throttling. In *International Symposium on Computer Architecture*, pages 298–309, June 2005.

[2] 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.

[3] 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):44–55, 2004.

[4] J. Corbalan, X. Martorell, and J. Labarta. Performance-Driven Processor Allocation. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):599–611, July 2005.

[5] J. Dorsey, S. Searles, M. Ciraula, E. Fang, S. Johnson, N. Bujanos, R. Kumar, D. Wu, M. Braganza, and S. Meyers. An Integrated Quad-Core Opteron(TM) Processor. In *IEEE International Solid-State Circuits Conference*, pages 102–103, February 2007.

[6] D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel Job Scheduling – A Status Report. In *Workshop on Job Scheduling Strategies for Parallel Processing*, June 2004.

[7] D. G. Feitelson. Job Scheduling in Multiprogrammed Parallel Systems. Technical Report RC 19790 (87657), IBM Research, August 1997.

[8] D. G. Feitelson and L. Rudolph. Metrics and Benchmarking for Parallel Job Scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–24, 1998.

[9] S. Ghiasi and D. Grunwald. Aide de Camp: Asymmetric Dual Core Design for Power and Energy Reduction. Technical Report CU-CS-964-03, The University of Colorado, Department of Computer Science, 2003.

[10] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of Both Latency and Throughput. In *International Conference on Computer Design*, pages 236–243, October 2004.

[11] T. Ibaraki and N. Katoh. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, 1988.

[12] 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.

[13] C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[14] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. In *International Symposium on Microarchitecture*, pages 381–394, December 2007.

[15] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *International Symposium on Microarchitecture*, pages 81–92, December 2003.

[16] 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.

[17] U. Nawathe, M. Hassan, K. Yen, L. Warriner, B. Upputuri, D. Greenhill, A. Kumar, and H. Park. An 8-Core 64-Thread 64b Power-Efficient SPARC SoC. In *IEEE International Solid-State Circuits Conference*, pages 108–109, February 2007.

[18] D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, January 2006.

[19] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and Exploiting Program Phases. *IEEE Micro*, 23(6):84–93, November/December 2003.

[20] 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.

[21] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. In *IEEE International Solid-State Circuits Conference*, pages 98–99, February 2007.

[22] 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.