Task Scheduling on Adaptive Multi-Core

Mihai Pricopi and Tulika Mitra

Abstract—Multi-cores have become ubiquitous both in the general-purpose computing and the embedded domain. The current technology trends show that the number of on-chip cores is rapidly increasing, while their complexity is decreasing due to power and thermal constraints. Increasing number of simple cores enable parallel applications benefit from abundant thread-level parallelism (TLP), while sequential fragments suffer from poor exploitation of instruction-level parallelism (ILP). Recent research has proposed adaptive multi-core architectures that are capable of coalescing simple physical cores to create more complex virtual cores so as to accelerate sequential code. Such adaptive architectures can seamlessly exploit both ILP and TLP. The goal of this paper is to quantitatively characterize the performance potential of adaptive multi-core architectures. Previous research have primarily focused on only sequential workload on adaptive multi-cores. We address a more realistic scenario where parallel and sequential applications co-exist on an adaptive multi-core platform. Scheduling tasks on adaptive architectures reveal challenging resource allocation problems for the existing schedulers. We construct offline and online schedulers that intelligently reconfigure and allocate the cores to the applications so as to minimize the overall makespan under the constraints of a realistic adaptive multi-core architecture. Experimental results reveal that adaptive multi-core architectures can substantially decrease the makespan compared to both static symmetric and asymmetric multi-core architectures.

Index Terms—Scheduling, adaptive multi-cores, dynamic heterogeneous multi-core, ILP, TLP, malleable and moldable tasks

1 Introduction

Computing systems have made the irreversible transition towards multi-core architectures due to power and thermal limits. Current generation processor architectures in both general-purpose computing and embedded domain are symmetric multi-cores consisting of a number of simple and identical cores (see Fig. 1). Such symmetric multi-core solutions are perfect match for easily parallelizable applications that can exploit significant thread-level parallelism (TLP). Indeed the current trend is to multiply the number of cores on chip to offer more TLP, while reducing the complexity of the cores to avoid power and thermal issues.

But most applications still comprise of a significant fraction of sequential workload. Amdahl's Law [18] states that such applications will suffer from limited instruction-level parallelism (ILP) exploitable in the simple cores. Single ISA (instruction-set architecture) but performance asymmetric multi-cores [21], comprising of both simple and complex cores, have recently been proposed as a promising alternative. The processors in an asymmetric multi-core architecture share the same ISA but their micro-architectures (pipeline and caches) are very different. Indeed, ARM has recently announced big.LITTLE processing [15] for mobile platforms where high-performance, triple-issue, out-of-order Cortex A-15 cores are integrated with energy-efficient in-order Cortex A-7 cores in the same chip.

Even though asymmetric multi-cores are positioned to accommodate software diversity (mix of ILP and TLP workload) better than symmetric multi-cores, they are not the ideal solution. As the mix of simple and complex cores has to be freezed at design time, an asymmetric multi-core lacks the flexibility to adjust itself to dynamic workload. The next logical step forward to support diverse and dynamic workload is to design a multi-core that can, at runtime, tailor itself according to the applications [19], [28], [29], [35] and thus create asymmetric configurations dynamically. Such adaptive architectures are physically fabricated as a set of simple, identical cores. At runtime, two or more such simple cores can be coalesced together to create a more complex virtual core (see Adaptive configuration in Fig. 1 with two virtual cores). Similarly, the simple cores participating in a complex virtual core can be disjoined at any point of time. A canonical example is to form coalition of two 2-way out-of-order (ooo) cores to create a single 4-way ooo core. Thus we can create asymmetric multi-cores through simple reconfiguration.

Adaptive multi-cores appear well poised to support diverse and dynamic workload consisting of a mix of ILP and TLP. As adaptive multi-core is a nascent area, existing research primarily focuses on developing appropriate micro-architectural techniques to form coalition of simple cores. The performance evaluation of the adaptive multi-core in these works [19], [28], [29], [35] only looks at how well a virtual complex core formed with simple physical cores can exploit ILP in a sequential application or TLP in parallel applications, independently. In reality, such adaptive architectures need to support both sequential and parallel applications executing concurrently. Existing literature is thus missing a realistic evaluation of the performance potential of adaptive multi-cores. In this paper, we take the first step towards filling this gap through a concrete performance limit study of adaptive multi-cores in a scenario where both parallel and sequential applications coexist in the system.

Conducting a limit study of adaptive architectures with realistic workload is a challenging problem. As we are

• *The authors are with the School of Computing, National University of Singapore, 117417 Singapore. E-mail: {mihai,tulika}@comp.nus.edu.sg.*

Manuscript received 20 Dec. 2012; revised 08 May 2013; accepted 12 May 2013. Date of publication 20 May 2013; date of current version 12 Sep. 2014. Recommended for acceptance by D. Bader.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.115

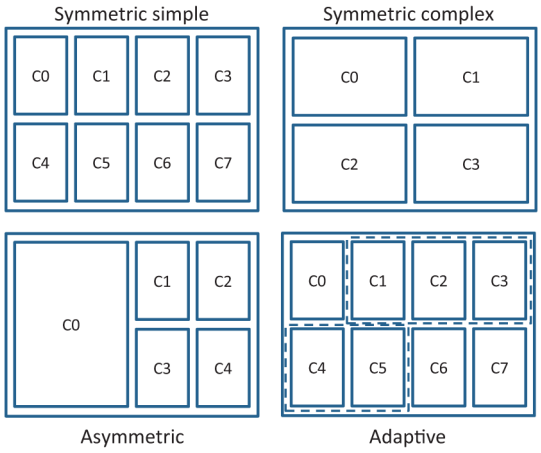


Fig. 1. Example of multi-core configurations.

interested in identifying the true performance potential of an adaptive multi-core architecture, we have to employ an optimal scheduler that can intelligently reconfigure and allocate the cores to the applications so as to minimize the makespan.

1.1 Illustrative Example

We present an example that provides a visual illustration of our scheduling problem. This example also concretely explains the challenges involved in conducting the performance limit study. For this example, we have chosen a set of five benchmarks: three sequential applications (*gobmk*, *quantum* and *fft*) from SPEC [1] and MiBench [17] benchmark suites that can exploit ILP through complex ooo cores, and two parallel applications (*bodytrack* and *blackscholes*) from PARSEC [3] benchmark suite that can exploit TLP through multiple simple cores. The experimental setup used to obtain the performance of each individual benchmark on different number of cores and configurations will be explained in Section 6.

We consider different multi-core architectures: (a) static symmetric architecture with eight 2-way ooo cores, (b) static asymmetric architecture with one 8-way ooo core and four 2-way ooo cores, (c) adaptive architecture with eight 2-way ooo cores where the cores can be coalesced together to form 4-way, 6-way, or 8-way complex core. We assume the area of a 8-way ooo core is roughly equivalent to that of four 2-way ooo cores; hence all the architectures are area-equivalent to the symmetric architecture with eight 2-way cores.

The Gantt charts presented in Fig. 2 show the schedules for different architectures along with the makespan (the time when all applications complete execution). The lower the makespan, the better is the performance of the system. For this example, we obtain the optimal schedule using the technique presented later in Section 4.

For the symmetric multi-core, the sequential applications are restricted to using only one core, while the parallel applications can benefit from multiple cores. This severely restricts the performance because the makespan of 500M cycles is defined by the sequential application *fft*.

The asymmetric multi-core architecture provides opportunity for the sequential applications to exploit ILP through one 8-way core. But the parallel applications are restricted to use only four simple cores. The sequential applications *fft* and *quantum* attempt to take advantage of the complex core to reduce execution time and reduce the demand for the simple cores. But this choice only leads to increased makespan. This example clearly shows that a fixed asymmetric solution may not always be the best replacement for the symmetric architecture due to the lack of flexibility and availability of the number of cores.

The adaptive multi-core architecture, on the other hand, carefully selects the number of cores allocated to each application. In this case, the sequential applications exploit ILP through core coalition, while the parallel applications exploit TLP by using multiple simple cores. In fact, the adaptive architecture dynamically creates five different asymmetric multi-core configurations during the makespan, in contrast to rigid symmetric and asymmetric solutions.

The example illustrates the challenges in forming the schedule for an adaptive architecture. For the optimal schedule in Fig. 2c, we have to first determine the allocation of the cores to the applications. This is a challenging task because an application can be allocated varying number of cores over time. For example, *fft* uses two cores for certain time interval and one core for the remaining time. Moreover, an application may be allowed to migrate from one core to another during its execution even if it uses fixed number of cores throughout execution. Finally, any realistic adaptive architecture imposes additional scheduling and allocation constraints that need to included in our decision process.

1.2 Contributions

We make the following concrete contributions in this paper.

- We perform a quantitative limit study of an ideal adaptive multi-core architecture for a mix of real sequential and parallel applications where the number of cores for both parallel and sequential applications can be adapted at runtime. We develop an *optimal scheduler for the ideal adaptive multi-core* that, given a mix of sequential and parallel tasks, can optimize the makespan via systematic reconfiguration and allocation of the cores to the tasks.
- We then develop a *scheduler for a realistic adaptive architecture*, namely Bahurupi [28], proposed recently. Bahurupi imposes additional scheduling and allocation constraints compared to the ideal adaptive architecture. Interestingly, it turns out that Bahurupi performs close to the optimal limit set by the ideal adaptive architecture and outperforms any static symmetric or asymmetric architecture across diverse workload.
- Finally, we design an online scheduler for adaptive multi-core architectures like Bahurupi that can be easily integrated in any contemporary operating system. The online scheduler allocates the cores to the tasks as they arrive. In this case also Bahurupi performs far better than both static symmetric and asymmetric multi-cores by successfully reconciling both ILP and TLP demands at runtime.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 presents a brief overview of a realistic adaptive multi-core architecture, Bahurupi, used in this study. Section 4 designs an optimal scheduler to evaluate the performance limit of an ideal adaptive architecture. This optimal schedule is then suitably modified to satisfy the constraints imposed by realistic Bahurupi architecture. An online scheduler for adaptive multi-core is presented in

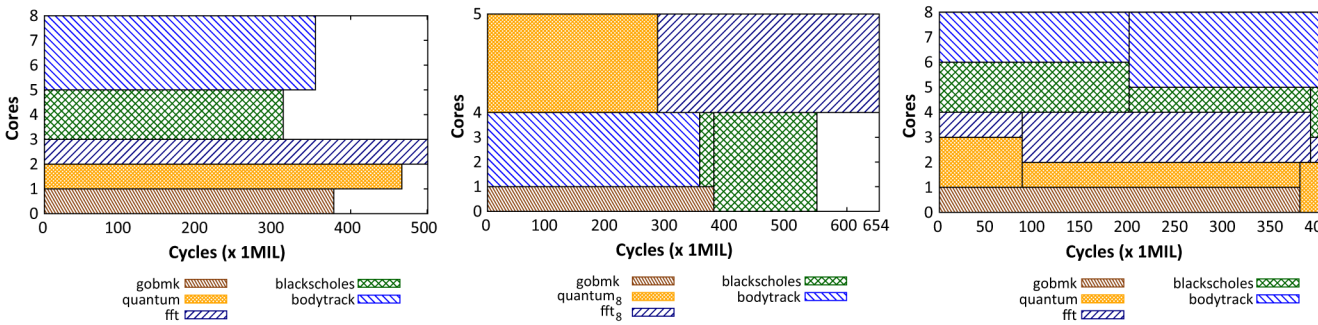


Fig. 2. Illustrative example showing the schedule of a mix of sequential and parallel applications on different architectures.

Section 5. The results of the quantitative evaluation appears in Section 6 and finally Section 7 concludes.

2 Related Work

A number of adaptive multi-core architectures have been proposed in the literature recently. Core Fusion [19] fuses homogeneous cores using complex hardware mechanism. They evaluate the performance by running serial tasks and parallel tasks separately on the adaptive multi-core. Voltron [35] exploits different forms of parallelism by coupling cores that together act as a VLIW processor. This architecture relies on a very complex compiler that must detect all forms of parallelism found in the sequential program. The evaluation is performed with only sequential applications and configuring the hardware to exploit different types of parallelism in the application. Federation [29] shows an alternative solution where two scalar cores are merged into a 2-way out-of-order (ooo) core. Again they evaluate the performance by running only sequential applications.

Bahurupi [28] is an adaptive architecture that offers a simple yet elegant approach through a hardware-software cooperative solution to form coalition of 2-way ooo cores that can behave as 4-way or 8-way ooo cores. In contrast to the other proposed adaptive multi-core architectures, Bahurupi can be easily implemented on top of existing multi-core processors, using existing ISA and limited amount of extra hardware that makes it power and energy efficient. [28] evaluates performance and energy advantages of this architecture using sequential applications. In our study, we choose Bahurupi as the baseline dynamic multi-core architecture on which we schedule both sequential and parallel benchmarks.

An analytic study is presented in [18] where Amdahl's law is adapted to different types of multi-core architectures. The study uses simplistic architecture and application models. In case of an application comprising sequential fragments, the results show that asymmetric multi-cores can offer potential speedups higher than symmetric solutions, while adaptive multi-cores are the best option, being able to offer speedups even higher than the asymmetric architectures. In contrast, our work uses a real dynamic architecture that runs a mix of real sequential and parallel benchmarks, and quantitatively shows the performance limits of such system taking into consideration all hardware and software constraints.

Scheduling only sequential applications on flexible multi-core architectures is studied in [16] where different algorithms for static and dynamic scheduling are proposed. However, this work is built on top of the TFlex adaptive architecture [20] that requires a special ISA (EDGE) configured to support distributed execution of sequential applications. The EDGE ISA adds significant overhead by bookkeeping the source code block structures and it relies on point-to-point communication to interchange data among cores. Moreover, [16] proposes an optimal static scheduling algorithm with a high time complexity, $O(nm^2)$, where n is the number of applications and m is the number of cores in the system.

In our work, we first model the applications as independent preemptive *malleable* tasks. Scheduling malleable tasks has recently received significant attention. Malleable tasks are parallel tasks that may be processed simultaneously by a number of cores, where the speedup of the task is dependent on the number of allocated cores. Malleable tasks are allowed to be preempted and change the number of cores during execution. Scheduling malleable tasks is a promising technique for gaining computational speedup when solving large scheduling problems on parallel and distributed computers [8], [34]. Real applications for malleable tasks have been presented among others in [2] for simulating molecular dynamics, in [12] for Cholesky factorization, in [4] for operational oceanography and in [5] for berth and quay allocation.

The malleable task model was first proposed in [31] and later studied in [23], [32], [25], and [30]. Scheduling independent malleable tasks without preemption is proved to be NP-hard [13] and related work on this topic focuses on finding sub-optimal solutions. [31] presented a polynomial λ-approximation algorithm for the malleable tasks problem starting from any λ-approximation algorithm for the 2D bin-packing problem. Following this work, [23] presented a two-approximation algorithm and [25] developed a heuristic with worst case performance guarantee of $\sqrt{3}$ that was later improved to $\frac{3}{2}$ in [26]. Scheduling malleable tasks on clusters of multi-cores is proposed in [14] where allocation of tasks to clusters is also considered.

Operating with malleable tasks presents significant challenges for online scheduling systems. In our work, we use a variation of the malleable task model—the *moldable* task model. Moldable tasks are parallel tasks that can be executed using an arbitrary number of cores; but they cannot change the core allocation during execution. The performance of a moldable task is directly related to the number of allocated cores. Suboptimal solutions for scheduling moldable tasks have been studied in [6], [7], and [11].

To the best of our knowledge, no previous work characterized the performance of an adaptive multi-core architecture when both sequential and parallel applications coexist in the system. Our work introduces the first performance limit study of adaptive multi-core architectures for this realistic scenario.

3 Bahurupi Adaptive Multi-Core

We select a recently proposed architecture called Bahurupi [28] as a representative adaptive multi-core architecture for this study. Our decision is influenced by the fact that Bahurupi offers a simple yet elegant approach towards core coalition through a hardware-software cooperative solution. It is a cluster based architecture that imposes realistic constraints on the scheduler.

Bahurupi architecture is built on top of a m-core symmetric multi-core architecture consisting of 2-way out-of-order cores. Fig. 3 shows an example of 8-core Bahurupi architecture with two clusters $(P_0 - P_3)$ and $(P_4 - P_7)$. In coalition mode, the cores can merge together to create a virtual core that can exploit more ILP. For example, a 2-core coalition behaves like a 4-way ooo processor, while a 4-core coalition behaves like a 8-way ooo processor. Bahurupi can only create coalition within a cluster. Thus each coalition can consist of at most 4 cores and each cluster can have at most one active coalition at a time. The highlighted cores in Fig. 3 are involved in two coalitions of two (P_0, P_1) and four $(P_4 - P_7)$ cores. In this example one parallel application runs its two threads on cores P_2 and P_3, one medium-ILP sequential application is scheduled to coalition $(P_0 - P_1)$ and one high-ILP sequential application is scheduled to coalition $(P_4 - P_7)$.

When running in coalition mode, participating cores cooperatively execute a single thread in a distributed fashion. Basically, the cores execute basic blocks of the sequential thread in parallel and fall back to a centralized unit for synchronization and dependency resolution. Dependency comes in the form of control flow and data dependence. Bahurupi handles these with compiler support and minimal additional hardware.

A new instruction called *sentinel instruction* is added to the ISA, which is the first instruction of each basic block of the code. Basic blocks are constructed at compile time along with the information about *live-in* and *live-out* registers to/from each basic block. This information is encoded in the corresponding sentinel instruction, which also embeds control flow information, specifically, length of the basic block and whether it ends with a branch instruction. Thus, sentinel instructions capture both the dependency and the control flow information among the basic blocks.

Physically, all the cores share a *global PC, synchronization logic, global register file*, and *global renaming logic*. Cores participating in the coalition make use of these global structures while other cores can independently run different threads.

The only communication across the cores is through the live-in and live-out register values. A broadcast mechanism is used to let the producer core send its live-out register value to any consumer core. Each core snoops the broadcast bus for live-in registers.

The architecture includes a cache structure with reconfigurable banked L1 instruction and data caches where each bank is associated with a core. Cores participating in the coalition share a combined L1 instruction and data cache reconfigured from their banks. L2 instruction and data caches are shared across all the cores irrespective of whether the core is in coalition or not.

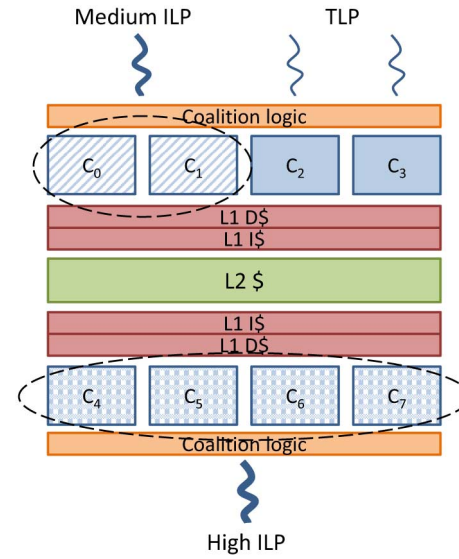


Fig. 3. Bahurupi adaptive multi-core architecture with 8 cores.

A key aspect of Bahurupi is its execution model. It emphasizes on operating in a distributed way with only a few essential global structures. This is important because any combination of cores can potentially become a coalition and the number of cores in the coalition does not change the way each core operates individually.

The overhead of adding/removing a core to/from a coalition is determined by the reconfiguration of L1 caches and internal resources. Bahurupi needs 100 cycles to reconfigure. This overhead is very small compared to the execution time of any application. The results presented in [28] show that, in case of integer applications, a 2-core or 4-core coalition can perform very close to a 4-way or 8-way out-of-order processor, respectively. On the other hand, for floating point applications, a 2-core or a 4-core coalition can even outperform a 4-way or 8-way true out-of-order processor. This is because Bahurupi can look far ahead in the future to exploit ILP as the compiler resolves dependencies across basic blocks. The reader may refer to [28] for details on high-level architecture of Bahurupi.

4 Optimal Schedule for Adaptive Multi-Core

The goal of this section is to conduct a quantitative performance limit study of the adaptive asymmetric multi-core architectures. We design an efficient off-line scheduler to carry out this limit study. We first present an optimal scheduler for an ideal adaptive multi-core architecture that is not restricted by any physical or technological constraint. Next, we impose additional constraints to perform scheduling on a realistic adaptive multi-core, namely Bahurupi (see Section 3).

4.1 Optimal Schedule on Ideal Adaptive Multi-Core

The ideal adaptive multi-core architecture consists of m physical 2-way superscalar out-of-order cores supporting shared

memory through hardware cache coherence. Any subset of these cores can be coalesced together to form one or more complex out-of-order cores. If r cores ($r \leq m$) form a coalition, then the resulting complex core supports $2r$-way superscalar out-of-order execution. The architecture can support any number of coalitions as long as the total number of cores included in all the coalitions at any point in time does not exceed m. We assume that the core coalition does not incur any performance overhead, that is, the performance of a $2r$-way core coalition is identical to a native $2r$-way core. A parallel application can execute on any subset of the simple cores, while a sequential application can execute on any simple core or a core coalition.

The adaptive architecture allows both sequential and parallel applications to use time varying number of cores. Thus we model the applications as *malleable workload* [22], where the number of cores allocated per application is not fixed and can change during execution through preemption. For the limit study, our goal is to create the optimal schedule for the malleable tasks[1] on the ideal adaptive architecture.

The scheduling problem can be formulated as follows. We consider an ideal adaptive multi-core architecture consisting of m homogeneous independent physical processors $\{P_0, P_2, \ldots, P_{m-1}\}$ running a set of n ($n \leq m$) preemptive *malleable tasks* $\{T_0, T_2, \ldots, T_{n-1}\}$. We assume that all the tasks arrive at time zero. The objective is to allocate and schedule the tasks on the cores so as to minimize the makespan $C_{max} = max_j\{C_j\}$ where C_j is the finish time of task T_j.

4.1.1 Optimal Schedule with Continuous Resources

We first determine the optimal C_{max} assuming that the number of cores allocated to a task need not be an integer. Then we transform this schedule to one that uses discrete resources.

Let us denote the number of processors assigned to a task T_j by r_j, where $0 < r_j \leq m$. If r_j is a continuous renewable resource (i.e., r_j can have real value), then we can adopt the solutions presented for the continuous resource allocation problem [33], [10] as our starting point.

Each task has a fixed amount of processing work $p_j > 0$. In a time interval of length t, a task performs $t \times g(t)$ amount of work where $g(t) = f_j(r_j) \geq 0$ is a continuous non-decreasing processing *speedup function* that relates r_j to the processing speed of a task. The set of *feasible resource allocations* of processors to tasks is as follows.

$$R = \left\{ \boldsymbol{r} = (r_0, \ldots, r_{n-1}) \,|\, r_j > 0, \sum_{j=0}^{n-1} r_j \leq m \right\}.$$

Applying speedup function $f_j(r_j)$ over the elements of R we obtain the set of *feasible transformed resource allocations*

$$U = \left\{ \boldsymbol{u} = (u_0, \ldots, u_{n-1}) \,|\, u_j = f_j(r_j), j = 0, \ldots, n-1, r_j \in R \right\}.$$

Theorem 1 (Resource Allocation Theory [33]). *Let $n \leq m$, convU be the convex hull of the set U, i.e, the set of all convex combinations of the elements of U, and $\boldsymbol{u} = \boldsymbol{p}/C$ be a straight line in the space of transformed resource allocations given by the parametric equations $u_j = p_j/C, j = 0, \ldots, n-1$. Then, the minimum schedule length is*

$$C^0_{max} = min\left\{ C \,|\, C > 0, \frac{\boldsymbol{p}}{C} \in convU \right\}, \quad (1)$$

where $\boldsymbol{p} = (p_0, \ldots, p_{n-1})$ is the processing work for the tasks.

From (1) it follows that, the minimum makespan value C^0_{max} is given by the intersection point of the line $\boldsymbol{u} = \boldsymbol{p}/C$ and the boundary of the convU set in the n-dimensional space of transformed resource allocations. The boundary of the convU set has a shape that depends on the convexity or concavity of the speedup functions f_j. The referred resource allocation solution is only valid for concave speedup functions f_j. In our evaluation, almost all the applications we tested can be approximated with concave speedup functions. In Fig. 4, we give a geometrical interpretation of the resource allocation problem applied in the case of $n = 2$ tasks, m processors and concave speedup functions f_0, f_1. The set of feasible allocations R is transformed into the set of feasible transformed allocations U by applying the speedup functions.

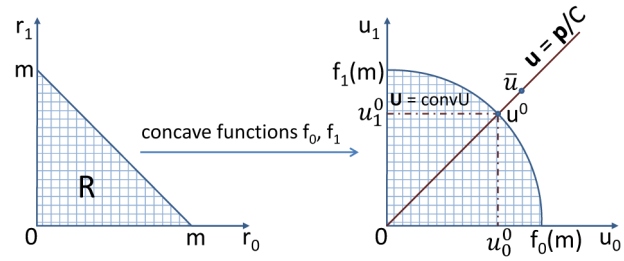


Fig. 4. Resource transformation example for $n = 2$.

The value of C^0_{max} is determined by the intersection point u^0, $C^0_{max} = p_j/u^0_j, j = 0, \ldots, n-1$. Thus, in order to find the minimum schedule length for our problem, we have to find the point u^0. [10] presents an algorithm that finds the solution for the continuous resource allocation problem in $O(n\, max\{m, nlog^2\, m\})$ time.

Normally, the speedup functions are only defined at integer points for a resource (discrete functions). The speedup functions f_j are extended with piecewise linear functions between consecutive r_j points. This way the monotonicity and concavity properties of the functions are maintained. The piecewise linear functions are described by the equations

$$\begin{aligned} &f_j(r) = b_{j,s} r + d_{j,s},\ r \in [s-1, s], \\ &s = 2, \ldots, m,\ j = 0, \ldots, n-1, \\ &b_{j,0} = d_{j,0} = 0. \end{aligned} \quad (2)$$

Fig. 5 shows an example of piecewise interpolation applied to the discrete speedup function for the *mcf* benchmark from SPEC benchmark suite with $m = 4$. Note that *mcf* is a sequential application. So the speedup on r cores correspond to the speedup on $2r$-way ooo core compared to the baseline 2-way ooo core.

4.1.2 Optimal Schedule with Discrete Resources

Clearly considering processors as continuous renewable resources is not a realistic assumption. Fortunately, the solution to the continuous problem can be transformed into a discrete solution with the same optimal makespan value C^0_{max}[9].

1. We use the terms *task* and *application* interchangeably.

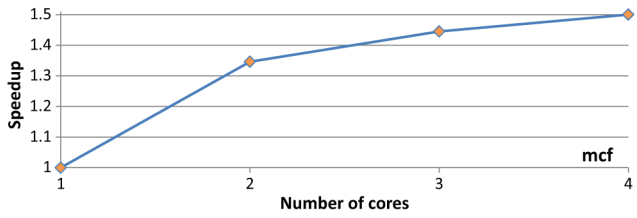


Fig. 5. Piecewise interpolation of speedup function for *mcf*.

The discrete solution is obtained from the continuous version through a rectangle packing procedure where two rectangles are allocated to each task. The rectangles represent the processing work and the number of cores allocated to a task. The dimensions (height and width) of the rectangles (a_j, v_j), (b_j, w_j) are computed as follows

$$\begin{aligned} a_j &= \lfloor r_j^0 \rfloor, b_j = \lceil r_j^0 \rceil, \\ v_j &= (b_j - r_j^0) p_j / f_j(r_j^0), \\ w_j &= (r_j^0 - a_j) p_j / f_j(r_j^0). \end{aligned} \tag{3}$$

Essentially, Equation 3 rounds the processor allocation r_j^0 up and down to integer values and represent r_j^0 as linear combinations of these two values. Consequently $b_j - a_j = 1$ for any task T_j. The rectangles are packed in (C_{max}^0, m) rectangle using the *rule of the southwest corner* in which a new rectangle is always assigned the leftmost position at the bottom of the unoccupied area. It can be proved [9] that the total width of the two rectangles can not exceed C_{max}^0. Also a rectangle always fits within the height m. Once a rectangle exceeds C_{max}^0 along the width, it is cut and the excess portion is moved back inside following the packing rule.

An example is shown in Fig. 6, where two rectangles of height b_j and a_j are allocated to application T_j. The rectangle of height b_j is packed first and it exceeds C_{max}^0. The highlighted part of the rectangle is moved back and then the second rectangle allocated for this task is placed. The packing algorithm guarantees at most two preemption points for each task and requires $O(n)$ time. Consequently, the time complexity to optimally schedule the malleable tasks on an ideal adaptive multi-core is $O(nmax\{m, n \log^2 m\})$.

Algorithm 1: Malleable Task Scheduler on Bahurupi

AdaptiveScheduler (*task_list*, m, n) **begin**
- *restart = FALSE;*
- ***Apply_constraint_C1***(***task_list***, **m**, **n**);
- *Find_contiunous_cmax*(*task_list*, m, n);
- *Convert_to_discrete*(*task_list*, m, n);
- *Generate_and_pack_rectangles*(*task_list*, m, n, *restart*);
- **if** *restart == TRUE* **then**
  - *AdaptiveScheduler*(*task_list*, m, n);
- ***Apply_constraint_C3***(*task_list*, **m**, **n**);

Generate_and_pack_rectangles (*task_list*, m, n, *restart*)

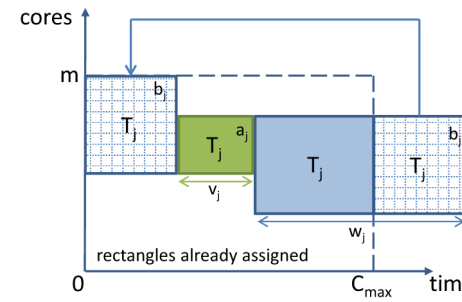


Fig. 6. Rectangle packing for discrete resource problem.

begin
- *constraint_violated = FALSE;*
- *current_task = 1;*
- *rectangles = Generate_rectangles*(m, n);
- *Order_rectangles*(*rectangles*, n);
- **while** *constraint_violated == FALSE* **do**
  - *Place_rectangles(rectangles[current_task]);*
  - ***Apply_constraint_C2(task_list, m, n, constraint_violated)***;
  - **if** *constraint_violated == TRUE* **then**
    - *restart = TRUE;*
    - *violating_task = current_task;*
    - *break;*
  - *current_task = current_task + 1;*
- **if** *constraint_violated == TRUE* **then**
  - *Constrain_remaining_tasks*(*current_task*, n);

Apply_constraint_C2 (*task_list*, m, n, *constraint_violated*)

begin
- **for** *all time intervals* (Δ_j, Δ_{j+1}) **do**
  - $n\_coalitions$ = *Count_coalitions*(Δ_j, Δ_{j+1});
  - CL_j = *Build_coalitions_list*(Δ_j, Δ_{j+1});
  - NCL_j = *Build_non_coalitions_list*(Δ_j, Δ_{j+1});
  - **if** $n\_coalitions > (m/4)$ **then**
    - *constraint_violated = TRUE;*
    - *break;*

Apply_constraint_C3 (*task_list*, m, n)

begin
- *constraint_violated = FALSE;*
- **for** *all tasks in* CL_j **do**
  - **for** $1 \leq k < (m/4)$ **do**
    - **if** *task uses cores* P_{4k} *and* P_{4k-1} **then**
      - *constraint_violated = TRUE;*
      - *break;*

if *constraint_violated* == *TRUE* **then**
  Pack_coalitions(CL_j);
  Pack_non_coalitions(NCL_j);

4.2 Task Scheduling on Bahurupi

When scheduling tasks on a realistic adaptive multi-core architecture, we must take into consideration all the constraints and limitations imposed by the system. More concretely, for Bahurupi architecture, we need to consider the following constraints in forming core coalitions for sequential tasks. The constraints are actually quite generic and are present in almost all adaptive multi-core architectures in the literature even though the exact values for the constraints can be different.

C1. A sequential application can use at most four cores.
C2. We can form at most $m/4$ coalitions at any time.
C3. A sequential application can only use cores that belong to the same cluster.

There is no such constraints for the parallel tasks. A parallel task may use any number of available cores to minimize the overall makespan. The scheduling solution for Bahurupi needs to add the constraints to the optimal scheduling solution presented in Section 4.1 for the ideal adaptive architecture. Algorithm 1 presents the scheduling algorithm for Bahurupi.

4.2.1 Constraint C1

Bahurupi restricts any sequential application to use at most four cores due to the limited amount of ILP found in sequential applications and the increase in coalition overhead when using more than four cores. To implement this constraint we modify the set of *feasible resource allocation* such that, the system can allocate at most four cores for sequential applications.

$$R = \left\{ r = (r_0, \ldots, r_j, \ldots, r_{n-1}) \,|\, r_j > 0, \sum_{j=0}^{n-1} r_j \leq m \right\},$$

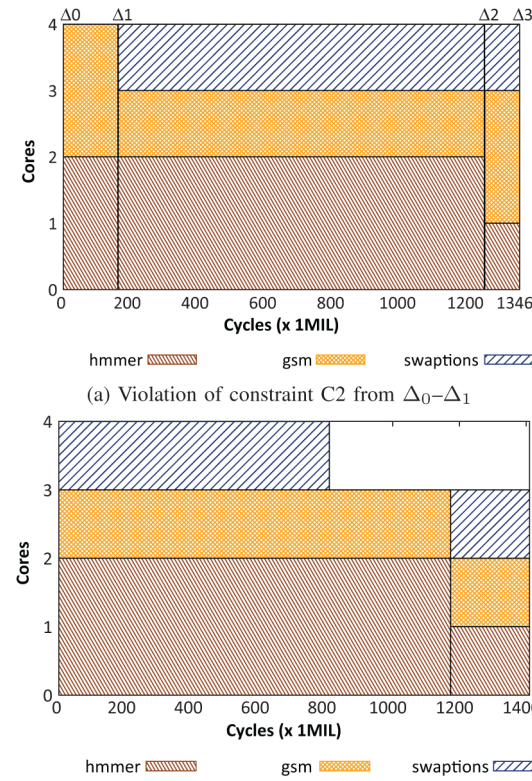
$$max(r_j) = \begin{cases} 4, & \text{if application } T_j \text{ is sequential,} \\ \text{m}, & \text{if application } T_j \text{ is parallel.} \end{cases}$$

Function *Apply_constraint_C1* in Algorithm 1 implements this constraint for the sequential tasks.

4.2.2 Constraint C2

Bahurupi architecture can accommodate at most one coalition per cluster. For each cluster, Bahurupi uses the coalition logic, which can be allocated to at most one coalition. Fig. 7 illustrates an example of a 4-core Bahurupi architecture running two sequential tasks, *hmmer* and *gsm* and one parallel task *swaptions*. This architecture can support at most one coalition. The time at which the tasks are preempted are marked on the top of the charts. The original schedule for the ideal adaptive architecture shown in Fig. 7a violates the bound on number of coalitions in the interval $\Delta_0 - \Delta_1$ where there are two coalitions of two cores ($\{P_0, P_1\}$ and $\{P_2, P_3\}$) used simultaneously by the tasks *hmmer* and *gsm*.

The one coalition per cluster constraint is imposed through $Apply\_constraint\_C2$ during the placement of the rectangles in the function *Generate_and_pack_rectangles* in Algorithm 1.



(a) Violation of constraint C2 from Δ_0–Δ_1

(b) Satisfying constraint C2 by restricting gsm to use one core

Fig. 7. Example of imposing constraint C2.

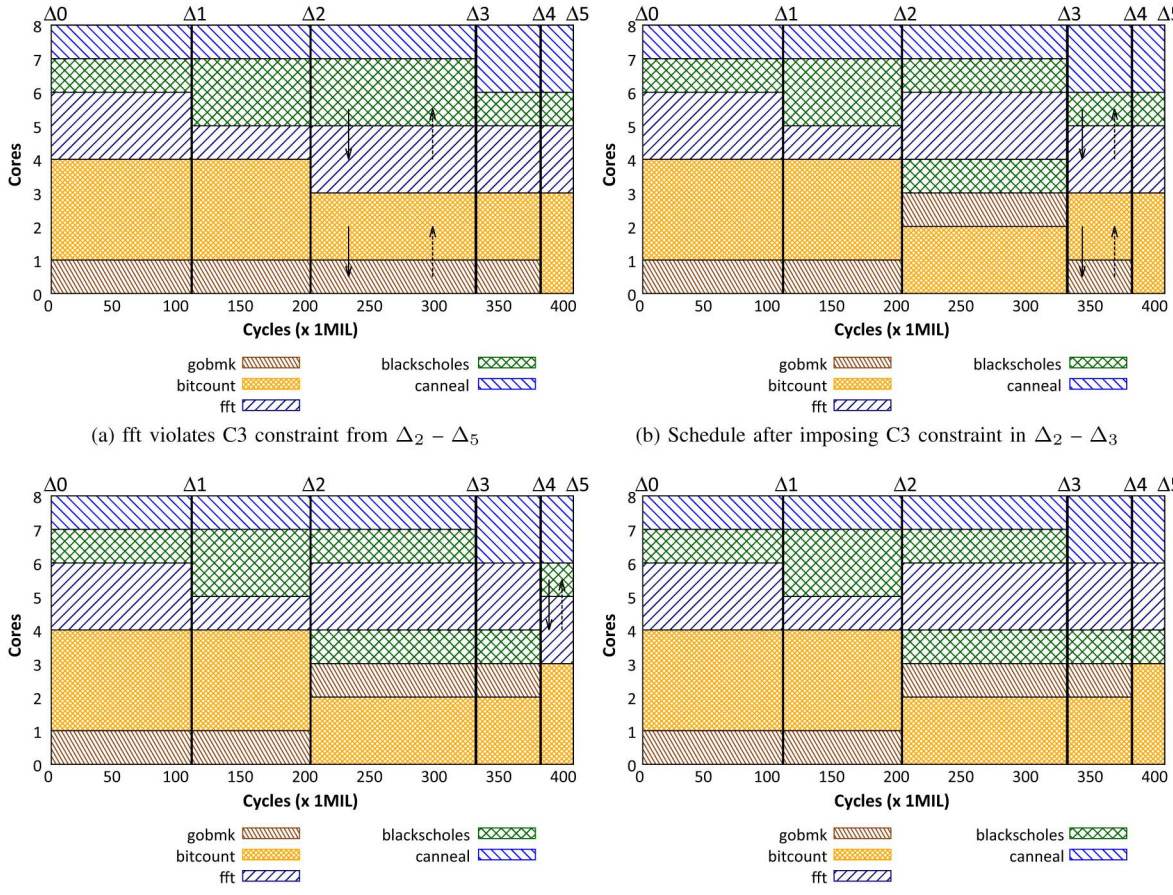
The rectangles are ordered initially by the function *Order_rectangles* in decreasing order of their heights. Rectangles having the same height are ordered in decreasing order of the corresponding processing work p_j. After placing a new rectangle, we scan each time interval ($\Delta_j - \Delta_{j+1}$) to count the number of coalitions used by the sequential tasks in that interval. If there are more than $m/4$ coalitions, then the constraint C2 is violated.

If the constraint C2 is violated, then we abort the rectangle packing and constrain the sequential tasks that are not placed yet (including the last placed task) to use only one processor, i.e., $r_j = 1$. The algorithm is then resumed with the new constraint. Imposing constraint C2 may lead to increased makespan. For example, in Fig. 7b, *gsm* is restricted to using only one core and the makespan is increased.

4.2.3 Constraint C3

Constraint C3 ensures that a sequential task is restricted to using only the cores within a cluster. In Fig. 8a we consider a set of three sequential applications *gobmk*, *bitcount* and *fft* and two parallel applications, *blackscholes* and *canneal*. We can see that the constraint C3 is violated for time intervals $\Delta_2 - \Delta_5$ assuming that processors $P_0 - P_3$ belong to one cluster and $P_4 - P_7$ belong to another cluster.

Function *Apply_constraint_C3* in Algorithm 1 implements this constraint. To impose this constraint, we first assume that the schedule has already been modified to satisfy the constraints C1 and C2. For each time interval $\Delta_j - \Delta_{j+1}$, we check if any sequential task uses cores across clusters. This can be done by simply checking if any sequential task uses cores P_{4k} and P_{4k-1}, where $1 \leq k < (m/4)$. If the constraint is violated in



(a) fft violates C3 constraint from $\Delta_2 - \Delta_5$

(b) Schedule after imposing C3 constraint in $\Delta_2 - \Delta_3$

(c) Schedule after imposing C3 constraint in $\Delta_3 - \Delta_4$

(d) Schedule after imposing C3 constraint in $\Delta_4 - \Delta_5$

Fig. 8. Example of imposing constraint C3.

any time interval, then we need to migrate the tasks within that interval.

For each time interval $\Delta_j - \Delta_{j+1}$, let CL_j be the list of rectangles for which the height is greater than 1 and the corresponding tasks are sequential (i.e., sequential tasks that need coalitions) and NCL_j be the list with rectangles that correspond to sequential tasks without coalition or parallel tasks. These lists are built while the rectangles are packed. As constraints C1 and C2 have already been satisfied, it follows that we can fit each rectangle from CL_j list on a unique cluster $P_{4k} - P_{4k+3}$, where $0 \leq k < (m/4)$. This is implemented by function *Pack_coalitions*. The one-unit height rectangles from NCL_j can fit in the available free cores regardless of the clusters, while the rectangles (threads) corresponding to the parallel tasks can be scheduled such that they fill the remaining free cores. This is done in function *Pack_non_coalitions*.

Fig. 8b shows the scheduling after applying the constraint C3 for the interval $\Delta_2 - \Delta_3$. Here, CL_2 = {*bitcount*, *fft*} and NCL_2 = {*canneal*, *blackscholes*, *gobmk*}. The tasks *bitcount* and *fft* are placed on cores $P_0 - P_1$ and $P_4 - P_5$ as they are sequential applications running on coalitions. After this step, the cores P_2 and P_3 are free. We choose *gobmk* to run on core P_2 and one thread of parallel application *blackscholes* to run on core P_3. Figs. 8c and 8d show the results for the rest of the intervals. As this step is only a rectangle rearrangement, application of the constraint C3 has no effect on the makespan.

Note that imposing constraints C1 and C3 maintain the optimality of the schedule. However, imposing constraint C2 may violate the optimality of the schedule. Among all the task sets we evaluated, only 1% of the task sets violate the constraint C2 in the optimal schedule on ideal adaptive multi-core. Thus, for most of the task sets, the schedule obtained for Bahurupi is the optimal schedule.

5 Online Schedule for Adaptive Multi-Core

The off-line schedule described in the previous section assumes all the tasks are ready at time zero. However, in a real system, the tasks can arrive at any point in time and the arrival times are not known beforehand. In this section, we present an online schedule for Bahurupi architecture to quantitatively evaluate the performance of an adaptive multi-core compared to static symmetric and asymmetric multi-cores.

Algorithm 2: Online scheduler for Bahurupi

InitAdaptiveOnlineScheduler (m, n) **begin**

    free_cores = m;

    free_clusters = m/4;

OnlineSchedule_tick()

```
begin
    if task_queue.empty() == FALSE then
        if free_cores > 0 then
            next_task = task_queue.front();
            Place_task(next_task);
        if current_task is finished then
            Update_free_cores(current_task, free_cores);
            free_clusters = free_cores/4;
Place_task (task)
begin
    max_cores = Get_max_cores(task);
    use_cores = max_cores;
    if max_cores < free_cores then
        use_cores = free_cores;
    if task.type == PARALLEL then
        Allocate_cores(task,use_cores);
    else
        use_cluster = (use_cores > 1);
        if (free_clusters-use_cluster) > 0 then
            Allocate_cores(task,use_cores);
            free_clusters = free_clusters-use_cluster;
        else
            use_cores = 1;
            Allocate_cores(task,use_cores);
    free_cores = free_cores-use_cores;
```

We allow the tasks to arrive in the system with different arrival times. Every task T_j is now defined by the tuple $<type_j,\ arr_j,\ f_{jk}>$, where $type_j$ is the type of the task (serial or parallel), arr_j is the arrival time and f_{jk} is its speedup function on k cores. The arrival times arr_j are randomly distributed in the interval $[0, \sum_{j=0}^{n-1} p_j]$ allowing the tasks to compete for limited number of free cores.

Algorithm 2 presents our online scheduler for Bahurupi adaptive multi-core. Here we model the workload as moldable tasks [22]. A *moldable task* can be scheduled on any number of cores just like malleable tasks but with the restriction that it cannot be preempted. This assumption makes it easy for us to integrate the scheduler in existing operating systems.

When a task arrives in the system, the only information required by the scheduler is its speedup function. This can be obtained by profiling the task on different number of cores. The scheduling decision is taken periodically at every system tick by the function *OnlineSchedule_tick*. As the moldable tasks cannot be preempted, once a task is scheduled on one or more cores using *Allocate_cores* function, it will run till completion. Once a task completes execution, the number of free cores is updated by *Update_free_cores* function.

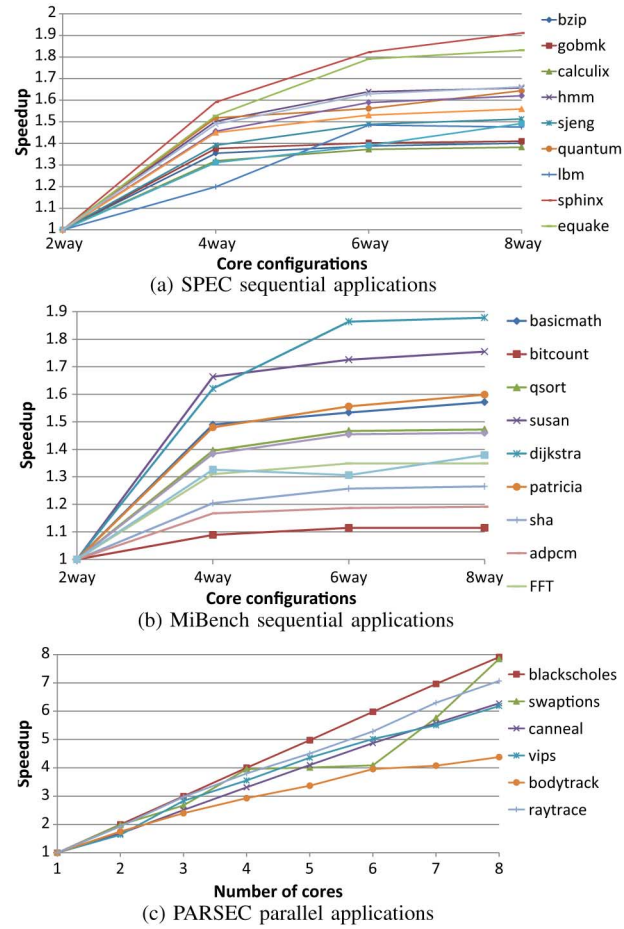


Fig. 9. Speedup functions for sequential and parallel tasks.

The dynamic allocation of the tasks to the cores is handled by *Place_task* function. For most tasks, the speedup function tends to have a flat region when applied to a large number of cores (see Fig. 9). In this region, the increase in performance on $r_j + 1$ cores is minimal compared to the performance on r_j cores. The function *Get_max_cores* returns the number of cores beyond which the speedup improvement is lower than 4%. This way we avoid allocating unnecessary cores that contribute little to performance improvement. Instead, these cores can be allocated to future tasks. The algorithm also ensures that the constraints C1, C2 and C3 mentioned earlier are satisfied. When constraint C3 is violated, function *Allocate_cores* migrates the tasks such that no sequential task spans across clusters.

6 Quantitative Results

In this section, we first present quantitative characterization of the performance limit of ideal adaptive multi-core and realistic adaptive multi-core (Bahurupi) compared to static symmetric and asymmetric multi-cores. This is followed by performance comparison of Bahurupi with static multi-cores using online scheduler.

6.1 Workload

We select 27 sequential applications from SPEC2006, SPEC2000 and embedded MiBench benchmark suites and 6 parallel applications from PARSEC benchmark suite. The characteristics of the benchmarks appear in Table 1.

TABLE 1
Benchmark Applications Used in Our Study

Benchmarks	Inputs	Suite	Type
gzip	input.source	SPEC2000	sequential
mesa	mesa.ppm		
mcf	inp.in		
equake	inp.in		
crafty	crafty.in		
ammp	ammp.in		
parser	test.in		
perlbmk	diffmail.pl		
bzip	input.program	SPEC2006	
gobmk	capture.tst		
calculix	beampic		
hmm	bombesin.hmm		
sjeng	test.txt		
quantum	50 5		
lbm	reference.dat		
sphinx	an4.ctl		
basicmath	runme_large.sh	MiBench	
bicount			
qsort			
susan			
dijkstra			
patricia			
sha			
adpcm			
fft			
gsm			
stringsearch			
blackscholes	simsmall	PARSEC	parallel
swaptions			
canneal			
vips			
bodytrack			
raytrace			

We generate different workload (task sets) consisting of varying mix of ILP and TLP tasks. We ensure that the tasks within a task set have similar processing workload so that all the tasks are competing for the resources throughout execution. This restriction in variability of processing workload is achieved as follows. Given the workload p_j for each task T_j, we compute the average workload and the standard deviation. We ensure that the ratio of standard deviation and average does not exceed 0.35 for a task set. Across all the tasks sets, the ratio of sequential tasks ranges from 25% to 80%; so the ratio of parallel tasks ranges from 20% to 75%.

6.2 Multi-Core Configurations

We model seven different static and adaptive multi-core configurations in our study as shown in Table 2. All these configurations are roughly area equivalent to eight 2-way multi-core architecture (S1) under the assumption that the area of a $2r$-way core is equivalent to that of r 2-way cores.

The symmetric eight 2-way multi-core architecture S1 is treated as the baseline. We also consider another symmetric multi-core S2 with medium complexity cores: four 4-way cores. The static asymmetric multi-core architectures A1, A2, and A3 employ different combination of small, medium, and large complexity cores. For example, A1 has more number of cores compared to A3 and hence is more suitable for TLP tasks, while A3 can accelerate ILP tasks better than A1.

The ideal adaptive multi-core and Bahurupi require the same amount of physical area as the baseline static symmetric multi-core. However, they can be morphed at runtime to form various different symmetric or asymmetric configurations. As mentioned earlier, the ideal adaptive architecture has no restriction on how core coalitions can be formed while Bahurupi imposes certain constraints driven by implementation considerations.

TABLE 2
Multi-Core Configurations Used in Our Study

Configuration	Description
(S1) 8x2-way	Symmetric eight 2-way cores
(S2) 4x4-way	Symmetric four 4-way cores
(A1) 2x4-way + 4x2-way	Asymmetric two 4-way + four 2-way cores
(A2) 1x8-way + 4x2-way	Asymmetric one 8-way + four 2-way cores
(A3) 1x8-way + 2x4-way	Asymmetric one 8-way + two 4-way cores
(Ideal) 8x2-way	Ideal adaptive multi-core
(Bahurupi) 8x2-way	Bahurupi Adaptive multi-core

We use MARSS cycle-accurate multi-core simulator [27] for our quantitative characterization work. The configuration parameters for out-of-order cores with different superscalarity values are shown in Table 3. The resources available to a core increases with increasing superscalarity.

6.3 Speedup Functions

Both static and adaptive multi-core architectures allow parallel tasks to use any number of cores. The speedup function for parallel tasks on different number of cores are shown in Fig. 9. We compile the parallel task with r threads and execute the threads on r cores to obtain the speedup. In other words, the speedup function represents the ideal scenario where the number of threads is equal to the number of cores. In reality, a parallel task compiled with r threads may need to use a different number of cores during execution. Similarly, for an adaptive architecture, a task can be allocated varying number of cores during execution. However, we noticed little difference in performance when an application compiled with m threads executed on r cores where $r < m$.

The static symmetric and asymmetric multi-cores use native 4-way and 8-way cores. The speedup of serial tasks on native 4-way and 8-way cores are obtained from MARSS cycle-accurate simulator [27]. Both ideal adaptive and Bahurupi architecture, on the other hand, employ core coalition to create virtual $2r$-way cores from r 2-way physical cores. We have established before [28] that the performance of a virtual $2r$-way core through core coalition in Bahurupi is either close to or even surpasses the performance of native $2r$-way core. For serial tasks running on virtual cores in adaptive architectures (Ideal and Bahurupi), we use speedup obtained from core coalition. As going beyond 8-way cores does not provide futher speedup due to limited ILP, we restrict the speedup function for serial tasks to 8-way core as shown in Fig. 9.

6.4 Scheduling on Static Multi-Cores

The scheduling algorithms presented in Section 4 are used to obtain the makespan for Ideal and Bahurupi architectures. For the symmetric architectures S1 and S2, we can employ the same optimal scheduling algorithm used for Ideal by simply restricting the sequential applications to use only one core.

For asymmetric architecture, however, we need to modify the scheduling algorithm. We first obtain the optimal makespan C^0_{max} assuming continuous resource allocation. Then for

TABLE 3
Configuration Parameters for Out-of-Order Cores: Issue, Commit, Dispatch Width; Reorder Buffer (ROB) Size; Load-Store Queue (LSQ) Size; Number of ALU, Floating Point (FP), and Load-Store (LSU) Units; Instruction-Data TLB Size, L1 Instruction-Data Cache Size, and Unified L2 Cache Size

Type	Issue width	Commit width	Dispatch width	ROB size	LSQ size	ALU cnt	FP cnt	LSU cnt	I/D TLB size	I/D L1$ size	I/D L2$ size
2-way	2	2	2	64	32	2	1	1	16	128K	2MB
4-way	4	4	4	128	64	3	2	2	32	256K	2MB
6-way	6	6	6	192	96	4	3	3	48	384K	2MB
8-way	8	8	8	256	128	5	4	4	64	512K	2MB

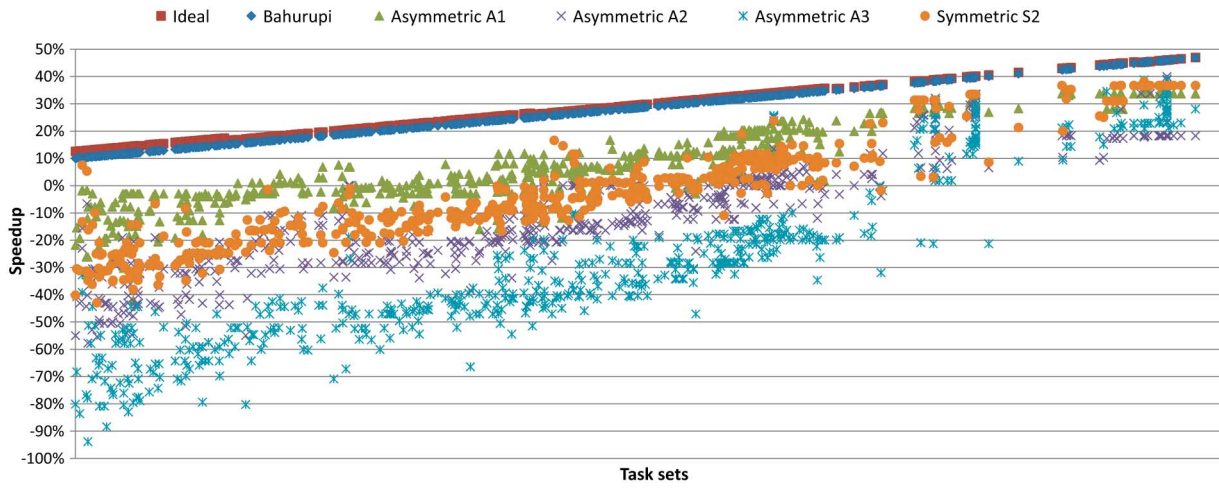


Fig. 10. Comparison of adaptive and static multi-cores under offline schedule. The speedup is w.r.t. symmetric multi-core S1.

each task T_j, we round up or down the resource allocated r_j^0 to match an available simple or complex core, except for $r_j^0 < 1$ in which case r_j^0 becomes 1. Finally, we perform strip packing [24] to optimally schedule the tasks. Note that scheduling using strip-packing is computationally expensive for static asymmetric multi-cores; but scheduling on static asymmetric multi-cores is not the focus of our work. We merely use it for comparison purposes.

For online schedule on static multi-cores, we adapt the online algorithm presented for Bahurupi in Section 5. In all the schedules, we assume a preemption penalty of 100 cycles and reconfiguration penalty of 100 cycles [28].

6.5 Limit Study of Adaptive Multi-Core

We now proceed to characterize the performance limit of adaptive multi-cores compared to static multi-cores. As mentioned earlier, we generate 850 task sets and compute the makespan of each task set on different multi-core architectures. The results are presented in Fig. 10. The speedup on Y-axis is defined w.r.t. the makespan on baseline symmetric S1 architecture consisting of eight 2-way cores. We plot the speedup on six different architectures for each task set represented on the X-axis. That is, each point in this graph represents the speedup of a particular task set on a particular architecture compared to baseline S1. For ease of presentation, the task sets along X-axis are sorted in non-decreasing order of speedup on Bahurupi.

The results clearly demonstrate that adaptive architectures (Ideal and Bahurupi) perform significantly better when compared to static symmetric and asymmetric architectures. It is interesting to note that the performance of Bahurupi is practically identical to that of Ideal adaptive architecture even though Bahurupi imposes certain constraints on core coalition. Thus, a cluster-based adaptive architecture like Bahurupi is quite effective in reaching the speedup limit set by ideal adaptive architecture.

The normalized speedup of adaptive architectures ranges from 10% to 49%. When the speedup on adaptive architecture is low, the speedup on static multi-cores is also very low or they perform even worse than the baseline symmetric architecture S1 due to lack of resources. On an average, adaptive architectures outperform the asymmetric configurations A1, A2 and A3 by 18%, 35% and 52% respectively. When compared with the symmetric configuration S2, the adaptive architecture performs 26% better, which makes the symmetric configuration S2 a better option than the asymmetric configurations A2 and A3. This is due to the availability of a number of powerful cores in S2 that can accelerate both the sequential and parallel tasks.

The results also anticipate the performance benefit of the announced asymmetric big.LITTLE multi-core [15], which includes two 3-way Cortex A-15 out-of-order cores and four dual-issue in-order Cortex A-7 cores on the same die. This

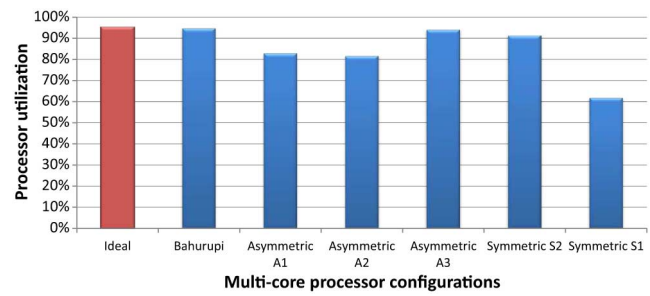


Fig. 11. Utilization of architectures in offline schedule.

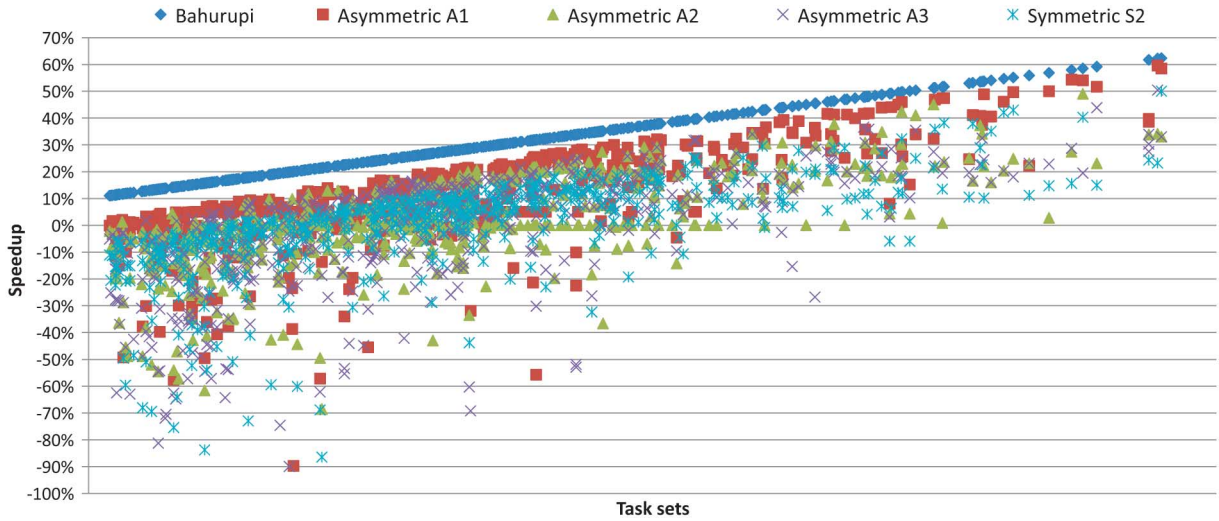


Fig. 12. Comparison of Bahurupi with static multi-cores under online schedule. The speedup is w.r.t. symmetric multi-core S1.

configuration is close to the asymmetric A1 configuration, which performs the best out of all asymmetric configurations.

Fig. 11 reports the processor utilization (averaged across all tasks sets) for different static and adaptive multi-core architectures. The adaptive architectures have the best utilization (94%) and performance making them the most efficient architectures. In contrast, the asymmetric multi-core A3 has a high utilization (92%) but low performance, making it the least efficient multi-core architecture. The symmetric configuration S1 has low utilization (61%) as it can only exploit TLP from parallel tasks. The serial tasks keep only a subset of the cores busy. The asymmetric configuration A1 has good utilization (82%) making it the most efficient asymmetric configuration.

6.6 Realistic Performance Benefit of Adaptive Multi-Core

We now present the performance benefit of adaptive multi-core architecture such as Bahurupi in the context of realistic online scheduling where tasks can arrive at arbitrary point in time. The results are shown in Fig. 12. The X-axis and Y-axis are defined similar to Fig. 10. We perform the experiments for the same 850 task sets used for offline schedule; however, we run each task set with five different arrival times to create a total of 4,250 task sets.

Again Bahurupi outperforms static symmetric and asymmetric multi-core architectures with speedup ranging from 10% to 62%. On an average, Bahurupi outperforms asymmetric A1, A2 and A3 configurations by 17%, 26%, and 28%, respectively. The symmetric architecture S2 shows a loss in performance of 26% when compared to Bahurupi. This shows that in the case of online scheduling, the speedup trend for different configurations is almost the same as that of off-line schedule (Fig. 10). The asymmetric configuration A1 offers the best asymmetric performance, while the configuration A3 offers the worst performance.

Fig. 13 plots the average processor utilization of the different architectures in online scheduling. The utilization trend is similar to that of offline schedule (Fig. 11). Bahurupi shows very good efficiency with 76% average utilization, while the asymmetric configuration A3 shows the worst efficiency with 67% average utilization. Similarly, symmetric configuration S1 has the lowest utilization (44%) due to the large number of simple cores that can only benefit the parallel applications, whereas the serial applications keep the occupied cores busy for a long time.

The measured average *competitive ratio* between our online scheduler and an optimal online scheduler (obtained using strip packing) is 1.14.

6.7 Reconciling ILP and TLP

The main objective of adaptive multi-core architectures such as Bahurupi is to reconcile the conflict between ILP and TLP tasks and provide performance benefit for both. We now provide quantitative validation that Bahurupi indeed manages to accelerate both sequential and parallel tasks.

As mentioned before, we use 27 sequential and 6 parallel applications to create 850 different task sets for our online scheduling experiment. Each task set is run with 5 different randomly selected arrival times to create a total of 850×5 different task sets. For each task and multi-core configuration used in our study, we compute the average speedup of the task across all the online schedules in which the task participates. The speedup is computed w.r.t. the execution time of the task on a single 2-way core.

Fig. 14 shows the speedup for the sequential applications. Bahurupi is the clear winner here and provides the best speedup for each application among all the multi-core configurations. Asymmetric A3 using native 8-way core is close to

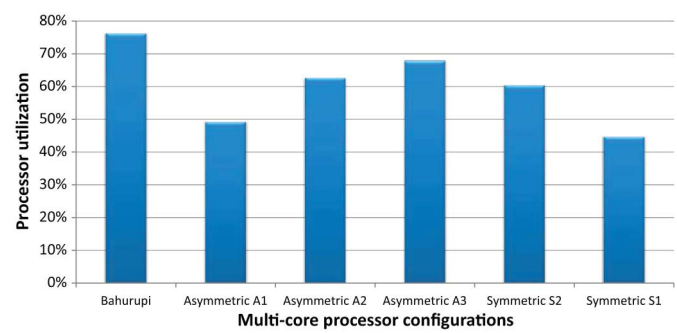


Fig. 13. Utilization of architectures in online schedule.

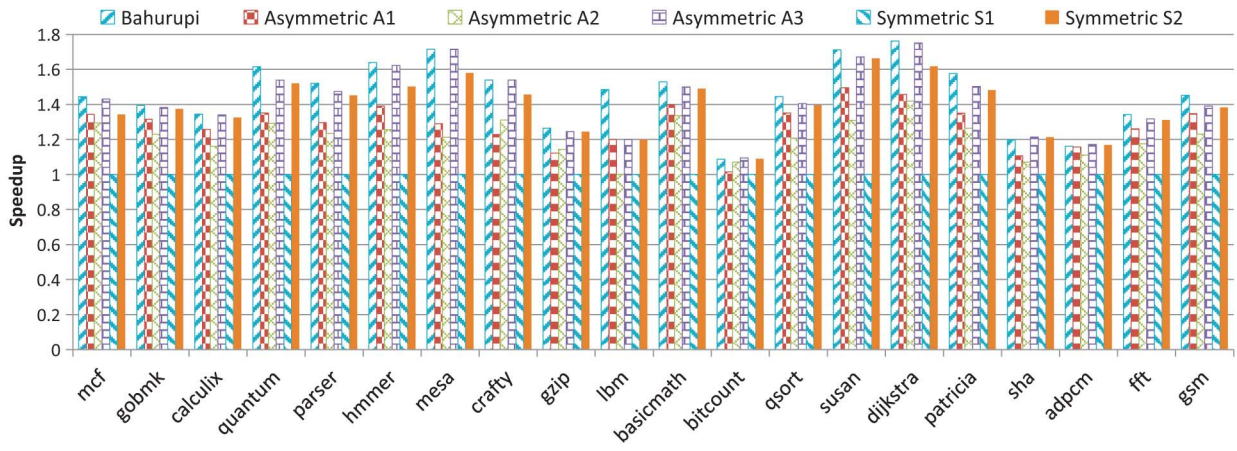


Fig. 14. Speedup of sequential applications averaged across all task sets normalized w.r.t. execution on native 2-way core.

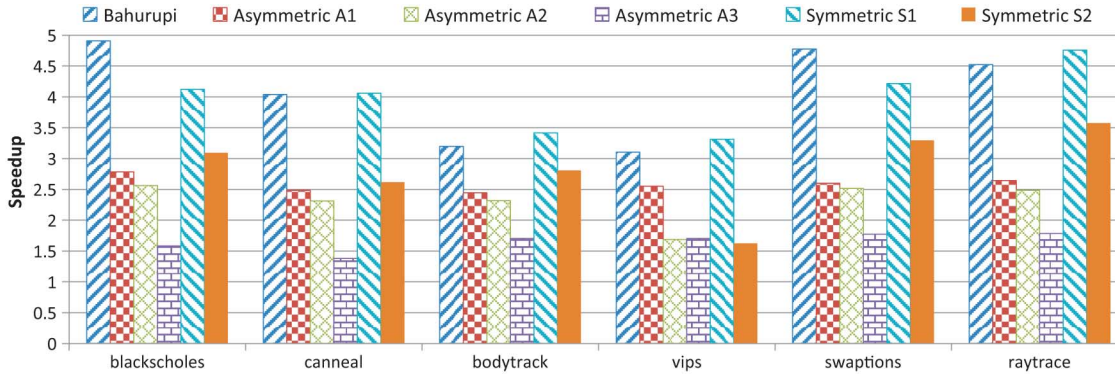


Fig. 15. Speedup of parallel applications averaged across all task sets normalized w.r.t. execution on one 2-way core.

Bahurupi. Also as expected, symmetric S2 deploys 4-way cores and hence has better speedup for serial tasks compared to symmetric S1 using 2-way cores. The performance of asymmetric A1 and A2 for serial applications appear somewhere in between.

In contrast, Fig. 15 shows the speedup for parallel applications. Again, the speedup of Bahurupi is close to that of baseline symmetric S1, which understandably has the best speedup because it has a large number of simple cores. The other symmetric and asymmetric configuration perform quite badly for parallel applications.

So in summary, adaptive multi-core architecture like Bahurupi is successful in accelerating both serial and parallel tasks. While symmetric S1 with large number of simple cores is quite effective for TLP, it shows poor performance for serial tasks. Asymmetric architecture A3 can perform well for serial tasks due to the presence of a complex core but suffers badly for parallel tasks. Among the static asymmetric configurations, the configuration A1 provides the best balance of ILP and TLP speedup; but is far behind adaptive multi-core architecture Bahurupi.

7 Conclusions

We have presented a comprehensive quantitative approach to establish the performance potential of adaptive multi-core architectures compared to static symmetric and asymmetric multi-cores. Ours is the first performance study that considers a mix of sequential and parallel workloads to observe the capability of adaptive multi-cores in exploiting both ILP and TLP. We employ an optimal algorithm that allocates and schedules the tasks on varying number of cores so as to minimize the makespan. This optimal schedule allows us to define the performance limit of ideal adaptive multi-cores for realistic workloads. We then modify this optimal schedule to satisfy the constraints imposed by a realistic adaptive multi-core, namely Bahurupi. The experiments reveal that both the ideal and the realistic adaptive architecture provide significant reduction in makespan for mixed workload compared to static symmetric and asymmetric architectures. Finally, we compare the performance of adaptive and static multi-cores in an online scheduling policy and demonstrate the same performance trend.

References

[1] *SPEC CPU Benchmarks*, http://www.spec.org/benchmarks.html, 2006.

[2] P.-E. Bernard, T. Gautier, and D. Trystram, "Large Scale Simulation of Parallel Molecular Dynamics," *Proc. 13th Int'l Symp. Parallel Processing 10th Symp. Parallel Distributed Processing*, pp. 638-644, 1999.

[3] C. Bienia, S. Kumar, J.P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Proc. 17th Int'l Conf. Parallel Architectures Compilation Techniques*, pp. 72-81, 2008.

[4] E. Blayo and L. Debreu, "Adaptive Mesh Refinement for Finite-Difference Ocean Models: First Experiments," *J. Physical Oceanography*, vol. 29, pp. 1239-1250, 1999.

[5] J. Blazewicz, T.C.E. Cheng, M. Machowiak, and C. Oguz, "Berth and Quay Crane Allocation: A Moldable Task Scheduling Model," *J. Operational Research Society*, vol. 62, no. 7, pp. 1189-1197, 2011.
[6] J. Blazewicz, M. Drabowski, and J. Weglarz, "Scheduling Multiprocessor Tasks to Minimize Schedule Length," *IEEE Trans. Computers*, vol. C-35, no. 5, pp. 389-393, May 1986.
[7] J. Blazewicz, M. Drozdowski, G. Schmidt, and D. de Werra, "Scheduling Independent Multiprocessor Tasks on a Uniform k-Processor System," *Parallel Computing*, vol. 20, no. 1, pp. 15-28, Jan. 1994.
[8] J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, *Handbook on Parallel and Distributed Processing*. Springer, 2000.
[9] J. Blazewicz, M.Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz, "Preemptable Malleable Task Scheduling Problem," *IEEE Trans. Computers*, vol. 55, no. 4, pp. 486-490, Apr. 2006.
[10] J. Blazewicz, M. Machowiak, J. Weglarz, M.Y. Kovalyov, and D. Trystram, "Scheduling Malleable Tasks on Parallel Processors to Minimize the Makespan," *Annals of Operations Research*, vol. 129, 2004.
[11] G.-I. Chen and T.-H. Lai, "Preemptive Scheduling of Independent Jobs on a Hypercube," *Information Processing Letters*, vol. 28, no. 4, pp. 201-206, July 1988.
[12] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst, *Numerical Linear Algebra on High-Performance Computers*. SIAM, 1999.
[13] J. Du and J.Y.-T. Leung, "Complexity of Scheduling Parallel Task Systems," *SIAM J. Discrete Math.*, vol. 2, no. 4, pp. 473-487, 1989.
[14] P.-F. Dutot and D. Trystram, "Scheduling on Hierarchical Clusters Using Malleable Tasks," *Proc. 13th Ann. ACM Symp. Parallel Algorithms Architectures*, pp. 199-208, 2001.
[15] P. Greenhalg, "Big. LITTLE Processing with ARM Cortex-A15 and Cortex-A7," technical report, ARM, 2011.
[16] D.P. Gulati, C. Kim, S. Sethumadhavan, S.W. Keckler, and D. Burger, "Multitasking Workload Scheduling on Flexible-Core Chip Multiprocessors," *Proc. 17th Int'l Conf. Parallel Architectures Compilation Techniques*, pp. 187-196, 2008.
[17] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "Mibench: A Free, Commercially Representative Embedded Benchmark Suite," *Proc. Workload Characterization*, pp. 3-14, 2001.
[18] M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33-38, July 2008.
[19] E. Ipek, M. Kirman, N. Kirman, and J.F. Martinez, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," *Proc. 34th Ann. Int'l Symp. Computer Architecture*, pp. 186-197, 2007.
[20] C. Kim, S. Sethumadhavan, M.S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S.W. Keckler, "Composable Lightweight Processors," *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, pp. 381-394, 2007.
[21] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," *Proc. 31st Ann. Int'l Symp. Computer Architecture*, p. 64, 2004.
[22] J.Y.-T. Leung, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* Chapman and Hall/CRC, 2004.
[23] W.T. Ludwig, "Algorithms for Scheduling Malleable and Non-Malleable Parallel Tasks," PhD thesis, Dept. of Computer Science, Univ. of Wisconsin-Madison, 1995.
[24] S. Martello, M. Monaci, and D. Vigo, "An Exact Approach to the Strip-Packing Problem," *INFORMS J. Computing*, vol. 15, no. 3, pp. 310-319, July 2003.
[25] G. Mounie, C. Rapine, and D. Trystram, "Efficient Approximation Algorithms for Scheduling Malleable Tasks," *Proc. 11th Ann. ACM Symp. Parallel Algorithms Architectures*, pp. 23-32, 1999.
[26] G. Mounie, C. Rapine, and D. Trystram, "A $\frac{3}{2}$-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks," *SIAM J. Computing*, vol. 37, no. 2, pp. 401-412, May 2007.
[27] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A Full System Simulator for Multicore x86 CPUs," *Proc. 48th Design Automation Conf.*, pp. 1050-1055, 2011.
[28] M. Pricopi and T. Mitra, "Bahurupi: A Polymorphic Heterogeneous Multi-Core Architecture," *ACM Trans. Architecture Code Optimization*, vol. 8, no. 4, pp. 22:1-22:21, Jan. 2012.
[29] D. Tarjan, M. Boyer, and K. Skadron, "Federation: Repurposing Scalar Cores for Out-of-Order Instruction Issue," *Proc 45th Ann. Design Automation Conf.*, pp. 772-775, 2008.
[30] D. Trystram, "Scheduling Parallel Applications Using Malleable Tasks on Clusters," *Proc. 15th Int'l Parallel Distributed Processing Symp.*, p. 199, 2001.
[31] J. Turek, J.L. Wolf, and P.S. Yu, "Approximate Algorithms Scheduling Parallelizable Tasks," *Proc. 4th Ann. ACM Symp. Parallel Algorithms Architectures*, pp. 323-332, 1992.
[32] Q. Wang and K.H. Cheng, "A Heuristic of Scheduling Parallel Tasks and Its Analysis," *SIAM J. Computing*, vol. 21, no. 2, pp. 281-294, Apr. 1992.
[33] J. Weglarz, "Modelling, and Control of Dynamic Resource Allocation Project Scheduling Systems," *Optimization and Control of Dynamic Operational Research Models*. S.G. Tzafestas, Ed., Elsevier, 1982.
[34] J.L. Wolf, P.S. Yu, J. Turek, and D.M. Dias, "A Parallel Hash Join Algorithm for Managing Data Skew," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1355-1371, Dec. 1993.
[35] H. Zhong, S.A. Lieberman, and S.A. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-Thread Applications," *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture*, pp. 25-36, 2007.

Mihai Pricopi received the bachelor's degree in computer engineering from the Faculty of Automatic Control and Computing Engineering of Iasi, Romania, and Master of Computer Engineering degree in advanced computing architectures, and later joined National University of Singapore in 2009. Currently, he is a PhD candidate working on dynamic heterogeneous computer architectures, static asymmetric architectures, and embedded systems. His major work is focused on the Bahurupi heterogeneous processor architecture that allows simple multi-cores to adapt dynamically creating more complex cores. He is also working on scheduling techniques for asymmetric and dynamic heterogeneous architectures that are able to exploit the energy efficiency of such systems.

Tulika Mitra received the BE degree from Jadavpur University, Kolkata, West Bengal, India, ME degree from Indian Institute of Science, Bangalore, Karnataka, India, and PhD degree from the State University of New York at Stony Brook, all in computer science in 1995, 1997, and 2000, resepctively. She is an associate professor in the School of Computing, National University of Singapore. Her research interest focuses on compilers and architectures for real-time embedded systems. Her work has received best paper award at International Conference of Field Programmable Technology 2012, best paper nominations at Design Automation Conference (2009 and 2012), International Conference of VLSI Design (2013), International Conference on Hardware/Software Codesign and System Synthesis (2008), International Conference on Field Programmable Logic and Applications (2007), and Euromicro Conference on Real-Time Systems (2007). She has served as a program committee member and associate editor of several leading conferences and journals in the embedded systems domain.