Parcae: A System for Flexible Parallel Execution

Arun Raman
Intel Research
Santa Clara, CA, USA
arun.a.raman@intel.com

Ayal Zaks
Intel Corporation
Haifa, Israel
ayal.zaks@intel.com

Jae W. Lee
Sungkyunkwan University
Suwon, Korea
jaewlee@skku.edu

David I. August
Princeton University
Princeton, NJ, USA
august@princeton.edu

Abstract

Workload, platform, and available resources constitute a parallel program's execution environment. Most parallelization efforts statically target an anticipated range of environments, but performance generally degrades outside that range. Existing approaches address this problem with dynamic tuning but do not optimize a multiprogrammed system holistically. Further, they either require manual programming effort or are limited to array-based data-parallel programs.

This paper presents Parcae, a generally applicable automatic system for platform-wide dynamic tuning. Parcae includes (i) the Nona compiler, which creates *flexible parallel programs* whose tasks can be efficiently reconfigured during execution; (ii) the Decima monitor, which measures resource availability and system performance to detect change in the environment; and (iii) the Morta executor, which cuts short the life of executing tasks, replacing them with other functionally equivalent tasks better suited to the current environment. Parallel programs made flexible by Parcae outperform original parallel implementations in many interesting scenarios.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors—Compilers, Run-time environments

General Terms Design, Performance

Keywords automatic parallelization, code generation, compiler, flexible, multicore, parallel, performance portability, run-time, adaptivity, tuning

1. Introduction

The emergence of general-purpose multicore processors has resulted in a spurt of activity in parallel programming models. Justifiably, the primary focus of these programming models has been parallelism extraction. However, parallelism extraction is just one part of the problem of synthesizing well-performing programs which execute efficiently in a variety of execution environments. The other, equally important part is the tuning of the extracted parallelism [29, 30]. In the absence of intelligent tuning, a parallel program may perform worse than the original sequential program [23, 40].

Parallel program performance depends on several environmental *run-time* factors. Synchronization and communication overheads are often difficult to predict and may erode the benefits of parallelism [40]. Parallel resources available to the program may vary, including number of cores and memory bandwidth [9, 23]. In addition, application workload can change during execution. Most parallel programs are produced by programmers or compilers with a single static parallelism configuration encoded at development or compile time. Any single program configuration is likely to become suboptimal with changes in the execution environment [33, 40].

Libraries such as OpenMP and Intel Threading Building Blocks allow programmers to encode multiple program configurations implicitly, and have an appropriate configuration chosen at run-time based on the execution environment [33, 40]. These libraries, however, require programmers to expend considerable efforts (re-) writing the parallel program. Furthermore, the run-time systems of these libraries optimize the execution of individual programs, without exploring platform-wide implications and opportunities.

Ideally, compilers should automatically generate flexible parallel programs which adapt to changes in their execution environment. Existing compiler and run-time systems tune parameters such as the degree of parallelism of data-parallel (DOALL) loops and block size of loops, either statically or dynamically, to match the execution environment [3, 5, 14, 17, 41, 44]. These systems are limited to optimizing array-based programs with communication-free data-parallelism, where the performance impact of those parameters can be relatively easily modeled. However, for general-purpose programs with complex dependency patterns, parallelism is typically non-uniform and involves explicit synchronization or communication. This significantly complicates performance modeling, often resulting in a large space of possibly effective parallelism configurations. Existing compiler-based parallelization algorithms for such general-purpose programs select a single configuration, typically one deemed most suitable for an unloaded platform [34, 43, 47].

This paper presents Parcae, a compiler and run-time software system that delivers performance portability for both array-based programs and general-purpose programs, extending the applicability of prior work. The Parcae compiler, Nona, creates flexible parallel programs by (i) extracting multiple types of parallelism from a program region; (ii) creating code to efficiently pause, reconfigure, and resume its execution; and (iii) inserting profiling hooks for a run-time system to monitor its performance. The Parcae run-time system, comprising the Decima monitor and the Morta executor, maximizes overall system performance by (i) monitoring program performance and system events, such as launches of new programs, to detect change in the execution environment, and (ii) determining the best configurations of all flexible parallel programs executing concurrently for the new environment.

We evaluate Parcae on two real platforms with 8 and 24 cores. Flexible parallel execution enabled by Parcae is compared with conventional parallel execution of programs generated by a compiler applying high quality parallelizing transforms: PS-DSWP [34, 43] and DOANY [31, 46], targeting 8 and 24 threads with OS load balancing. Across seven benchmarks, Parcae reduces execution time by -1.4% (a small slowdown) to 41.9% with concomitant estimated processor energy savings of 23.9% to 83.9%. Platform-wide, Parcae reduces execution time of an entire workload by 12.8% to 53.5% with concomitant estimated processor energy savings of 22.4% to 76.3%.

In summary, the main contributions of this work are:

1. a compilation framework that is sufficiently powerful to automatically transform both array-based and general-purpose sequential programs into flexible parallel programs (Section 3),
2. a lightweight monitoring and execution framework to determine and enforce platform-wide optimal parallelism configurations of multiple concurrently executing programs (Section 4), and
3. a prototype implementation and its evaluation on seven benchmarks demonstrating its effectiveness in delivering robust portable performance (Section 6).

2. Run-time Variability and Implications

During program execution, the workload processed by the program may change phase or the parallel resources available to the program may vary. As an example, consider the problem of determining an ideal program configuration while the number of cores allocated to a program by the OS varies. This is a common scenario in platforms that are shared by multiple programs. Figure 1 shows optimized performance of the `blackscholes` option pricing benchmark from the PARSEC benchmark suite [7]. The compiler parallelized the outermost loop using a three-stage pipeline parallelization scheme (sequential-parallel-sequential). The program was executed on two different platforms, processing three different reference inputs, and making use of different number of cores. Observe that:

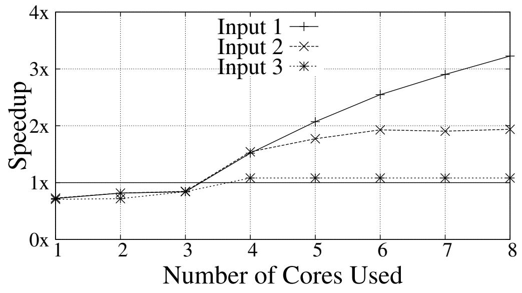
- On both platforms, the parallel version performs worse than sequential if three cores or fewer are employed.
- On Platform 1, the parallel version scales to 8 cores on Input 1, plateaus at 6 cores on Input 2, and is only slightly better than sequential on Input 3.
- On Platform 2, the parallel version scales to 10 cores but degrades afterwards on Input 1, plateaus at 4 cores on Input 2 (with 34% speedup), and performs worse than sequential on Input 3.

The reason for poor performance on Input 3 is the significantly greater communication bandwidth requirement, which overwhelms the benefits of parallelization. From the example, we infer that:

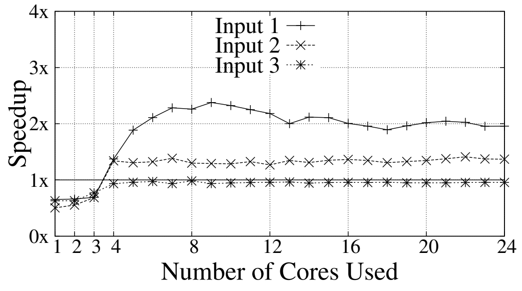
- The optimal configuration depends in general on both platform and workload characteristics.
- Employing more cores may not result in better performance, and may even degrade it. Moreover, parallel versions may even be slower than the sequential version.
- If an optimal configuration uses fewer cores than the total available, energy can be saved by switching off unused cores.
- The range of scenarios and varying concerns lay undue burden on programmers to ensure performance portability.

This work presents a system that optimizes and executes programs in a way that flexibly adapts them to new execution environments. Based on the above inferences, the proposed system:

1. prepares multiple parallel versions for each parallel region, as well as a sequential baseline version;
2. optimizes thread-level-parallelism leveraging both data- and pipeline-parallelism schemes;
3. generates parallel code that can be efficiently tuned and replaced online as the parallel region executes;
4. periodically optimizes and tunes multiple simultaneously executing programs for the duration of their execution; and



(a) Platform 1: 8-core, 8 GB RAM



(b) Platform 2: 24-core, 24 GB RAM

Figure 1. Performance variability running `blackscholes` option pricing across 3 workloads and 2 parallel platforms.

5. performs all of the above automatically, easing the burden on the programmer.

Figure 2 shows an example of the proposed execution model on a hypothetical five-core machine. Each parallel region consists of a set of concurrently executing tasks. (The inscription inside each box indicates the task and iteration number of a region; e.g., $M5$ represents the fifth iteration of task M.) At time t_0, program P_1 is launched with a pipeline parallel configuration (PS-DSWP[1]) having three stages corresponding to tasks A, B, and C. A and C are executed sequentially whereas B is executed in parallel by 3 cores as determined by the run-time system. At time t_1, another program P_2 is launched on the same machine. In response, the run-time system signals P_1 to pause at the end of its current iteration (iteration 5). The core receiving this signal (Core 1) acknowledges the signal at time t_2 and propagates the pause signal to the other cores. At time t_3, P_1 reaches a known consistent state, following which the run-time system determines a new allocation of resources to programs P_1 and P_2, say 2 cores to P_1 and 3 cores to P_2. At time t_4, the run-time system launches DOANY[2] execution of both programs P_1 and P_2. For P_1, task K and the synchronization block implement the same functionality as tasks A, B, and C.

3. Compilation for Flexible Parallel Execution

The Nona compiler identifies parallelizable regions in a sequential program and applies multiple parallelizing transforms to each region, generating multiple versions of *flexible code*. The generated flexible code can be paused during its sequential or parallel execution, reconfigured, and efficiently resumed by the Morta task executor. Nona also inserts profiling hooks into the generated code for the Decima monitor to observe program behavior. The parallelizing

[1] The PS-DSWP transformation splits a loop body across stages and schedules them for concurrent execution. It enforces dependencies through inter-stage communication channels.

[2] The DOANY transformation schedules loop iterations for parallel execution while synchronizing shared data accesses by means of critical sections.

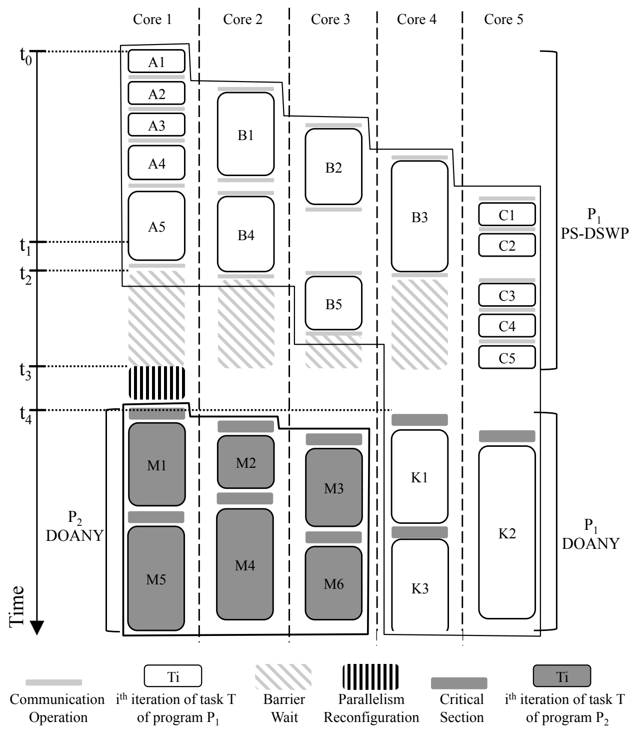


Figure 2. Parcae execution model: (t_0) program P_1 is launched; (t_1) program P_2 is launched; (t_2) P_1 acknowledges signal to pause; (t_3) P_1 reaches a known consistent state; (t_4) new resource allocation is determined and parallel execution of P_2 begins while P_1 switches to a parallelization that is better for two cores.

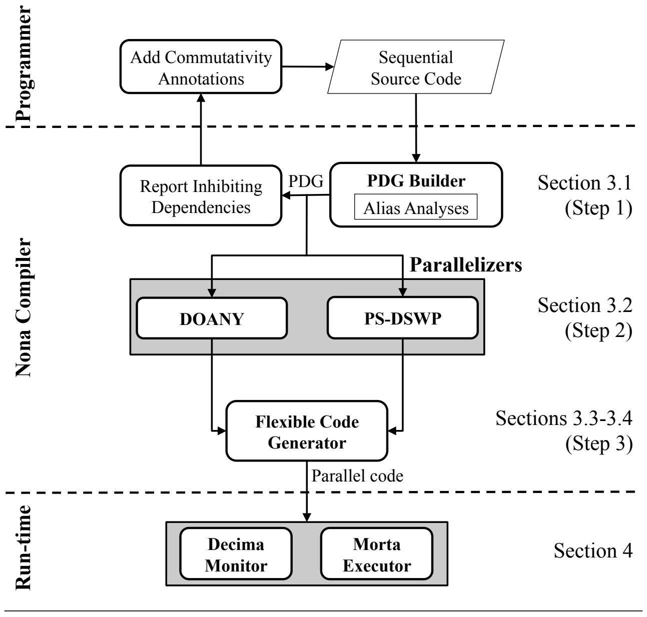


Figure 3. Parcae Architecture and Parallelization Workflow

transforms which we focus on in this work target loop nests as their candidate regions. We therefore describe the generation of flexible code at this level, implemented in the following three steps.

3.1 Parallelism Discovery and PDG Building

In the first step, Nona discovers parallelism by building a Program Dependence Graph (PDG) of the hottest outermost loop nest [1]. Some data dependency edges in the PDG can be relaxed and their corresponding nodes allowed to execute in parallel by applying special techniques such as privatization and re-association of reduction operations. Nona automatically identifies min, max, and sum reductions [1]. Other dependency edges might be relaxed, allowing the corresponding nodes to execute in either order, but not concurrently; Nona processes commutativity annotations provided by the programmer for this purpose [31, 43]. As Figure 3 shows, Nona propagates these annotations from the source code to the PDG, relaxes dependencies between commutative operations, and synthesizes the appropriate synchronization to ensure atomicity.

3.2 Multiple Parallelizations

In the second step, Nona applies multiple parallelizing transforms to the PDG of a loop nest. In this work, we employ a data-parallel transform with critical sections (DOANY [31, 46]) and a pipeline transform (PS-DSWP [34, 43]). The framework can accommodate additional, new transforms. Each transform extracts a distinct form of thread-level parallelism encapsulated in code packages called *tasks*. The original, sequential version of the loop is also maintained as a task.

Each task essentially contains the body of a loop, whose iterations must either execute sequentially or may execute in parallel. A task is labeled as either *sequential* or *parallel* accordingly. Note that parallel execution may involve communication or synchronization. A dynamic *instance* of a task refers to any single iteration of the loop contained in the task. Dynamic instances of a sequential task cannot execute concurrently with each other, whereas dynamic instances of a parallel task can.

DOANY tests the PDG with commutativity relaxations produced in the first step for absence of loop-carried dependencies to determine applicability. If there are no loop-carried dependencies, DOANY extracts the loop body into a parallel task, marshals all loop live-ins into the task via the heap, and inserts appropriate synchronization operations to ensure atomicity of all commutative operations. Iterations of the loop thus constructed may be executed concurrently by multiple threads without violating program semantics.

PS-DSWP is a widely applicable parallelization technique that partitions the loop into "stages", with each stage comprised of groups of instructions that are cyclically dependent on each other. Each stage essentially corresponds to a loop containing part of the original loop's body. These stages are constructed by building the strongly connected components (SCCs) of the PDG and partitioning the induced SCC graph. Inter-stage dependencies are satisfied via appropriate communication. A stage is sequential or parallel depending on whether the intra-stage dependencies are carried by the transformed loop's backedge or not. Iterations of a parallel stage may be executed concurrently by multiple threads.

The tasks extracted by each parallelizing transform are initially generated by the Multi-Threaded Code Generation algorithm (MTCG) [34]. MTCG operates in four steps (see Figure 4(a)). First, for each task, MTCG generates a new control flow graph (CFG) containing the relevant basic blocks. Second, MTCG inserts instructions corresponding to each basic block in the newly created CFG for that task. Third, MTCG inserts inter-task communication instructions (*send* and *recv* in case of PS-DSWP) and synchronization instructions (*lock* and *unlock* in case of DOANY). Fourth, MTCG replicates branch instructions into the newly created CFGs as necessary to match the original CFG.

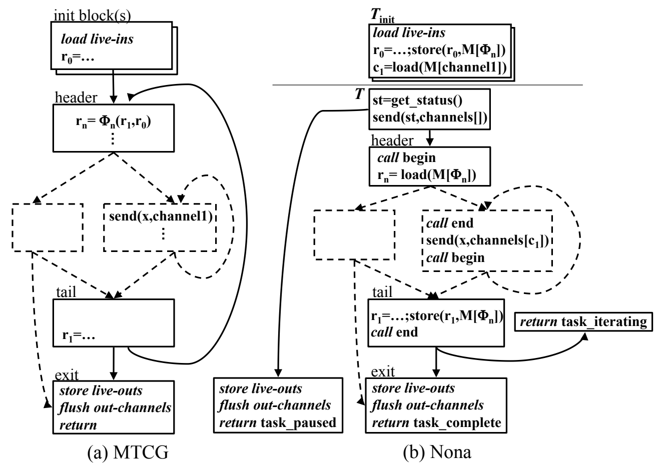


Figure 4. Original and transformed task code produced by MTCG and Nona, respectively. Dashed blocks/arcs represent arbitrary control flow within the loop.

Algorithm 1: Control logic for executing task instances

```
// getTaskInstance() blocks until reconfiguration ends or until next
   region begins. It returns NULL when program ends.
while instance ← runtime.getTaskInstance() do
    retVal ← invoke(instance.function, instance.args);
    if retVal == task_iterating then
        taskIterCount[instance.getTaskID()]++;
    else
        // retVal == task_paused or task_complete
        instance.flushOutChannels();
        region ← instance.getRegion();
        region.waitOnBarrier();
        if retVal == task_complete then
            region.terminate()
```

3.3 Flexible Code Generation

In the third step, Nona adapts the code generated by MTCG for flexible execution. Originally, the code generated by MTCG targets a fixed number of statically bound threads for each task. However, flexible parallel execution entails dynamic scheduling of task instances across different threads, execution of parallel tasks by a varying number of threads, and pausing a set of tasks followed by resumption of a possibly different set of tasks. Flexible parallel execution is hindered by:

1. Dependencies through private storage: A thread's registers and stack are private. Consequently, cross-iteration dependencies (between one task instance and another) carried by registers and the stack inhibit executing instances across multiple threads.
2. Inter-task communication: MTCG constructs point-to-point communication channels between threads executing tasks, and communicates dependencies in round-robin order across threads. If the number of threads executing each task varies at run-time, the dependencies across the communicating tasks may be reordered, violating sequential semantics.
3. Cross-iteration dependencies: These also inhibit the pausing of executing tasks and relaying of work remaining in the parallel region to a new set of tasks.

To facilitate flexible execution, Nona applies the following changes:

1. Upon completing each iteration, every task yields to the runtime system that determines whether the task should pause or may resume execution on the same thread or a different thread.
2. Registers and stack variables that are live across iterations of sequential tasks are saved and reloaded on the heap at the end and beginning of each iteration, respectively.
3. Parallel tasks avoid having local state across iterations, by sharing cross-iteration data in global memory .
4. When pipelined tasks pause, they flush their communication channels, sending all pending items down the pipeline.
5. Upon resumption, tasks execute an initialization sequence to reload invariant live-in data, and the run-time system resets the communication channels.

3.3.1 Changes to Task Control Flow

Consider the control flow graph CFG$_T$ of an arbitrary task T, as generated by MTCG (Figure 4(a)). CFG$_T$ represents a single-entry-single-exit code region containing a loop with a single `tail`→`header` backedge. There is a single entry edge reaching `header` block from outside the loop; there may be multiple exits from the loop, but all reach a single `exit` block which cannot be reached from outside the loop.

Nona modifies CFG$_T$ to support migration (see Figure 4(b)). The backedge is redirected from `tail` to a new exit block which returns `task_iterating` (instead of reaching the `header`). The original `exit` block now returns `task_complete`. Section 3.3.4 discusses the third (and last) exit block which is reached from a new pre-header block and returns `task_paused`.

The control logic to execute task instances is extracted into a separate loop, shown in Algorithm 1. The Morta executor sets up every worker thread to execute this loop. Upon receiving `task_iterating`, the thread increments a counter that tracks the number of iterations per task. Upon completing or pausing a task, the thread waits for other tasks of the region to complete or pause by means of a barrier, before starting to execute a new task.

3.3.2 Saving and Restoring State

Sequential tasks may have cross-iteration dependencies that flow through registers and variables on the stack, which are local to a thread. To facilitate lightweight migration to another thread, Nona inserts code to copy such variables to the heap at the end of each iteration and reload them at the beginning of each iteration. Note that the amount of information that needs to be copied through the heap is typically much smaller when applied between iterations, than at arbitrary locations as in general context switches or checkpoints.

Nona uses Static Single Assignment form (SSA) to represent code. In SSA form, loop-carried register dependencies are captured by ϕ nodes in the loop `header`. Figure 4 shows how flows through registers are converted into flows through the heap. Note that a register value need only be stored to the heap at the end of an iteration, not on every write to the register. This minimizes the cost of saving register state. A similar treatment addresses stack variables. The figure also shows how blocks preceding the loop header are extracted into a separate function (T_{init}). This function includes the loading of loop-invariant live-in values, and will be executed at every task activation and resumption.

3.3.3 Inter-task Communication

In PS-DSWP parallelizations, dependencies between tasks executing different stages need to be enforced. MTCG communicates such dependencies by inserting instructions for *send-receive* operations over point-to-point communication channels [34].

Communication between two sequential tasks is straightforward; communication between a sequential task (S) and a parallel task (P) involves data arbitration and merge. Consider a dependency edge $u \to v$ where $u \in S$ and $v \in P$. Let p be the (varying) number of threads that execute P. The value produced by the thread executing S flows to each of the threads executing P in a round-robin fashion: on the i^{th} instance of S, the value is communicated through the $(i \bmod p)^{th}$ channel. MTCG uses the induction variable (i), which is incremented once per instance of S, to identify the channel for a value that flows over the dependency edge. This holds analogously for dependency edges from a parallel task to a sequential task.

In MTCG, the number of threads p that execute a parallel task P is fixed at compile-time. This is not the case for Nona because the value of p may be changed by Mortaduring execution. Still, the above communication mechanism suffices provided p and the communication channels are maintained as run-time parameters, and that task instances are made *relayable*.

3.3.4 Relayable Task Instances

As described in Section 3.3.2, parallel tasks are constructed to have no local cross-iteration dependencies, and sequential tasks have all their cross-iteration data placed in the global heap between iterations. Thus, every task instance can halt after finishing or before starting an iteration, leaving the program in a known consistent state. Once instances halt, the system can be reconfigured safely and subsequent instances will continue to execute according to the new set of tasks and their thread allocation.

At the beginning of each iteration, every instance checks for a pause signal (see `get_status()` in Figure 4(b)), received either directly from Morta or from another instance. If an instance receives a pause signal, it propagates the signal and yields to Morta by returning `task_paused`. These signals are initiated and propagated as follows.

Initiating pause signals: When Morta chooses to reconfigure the program, it sends a pause signal to designated *master* tasks. In the DOANY and sequential versions, the single task is the master task, and in the case of PS-DSWP, only the task executing the first stage is designated as master. The master tasks are designed to query Morta for pause signals at the beginning of each instance, to which Morta responds with either `continue` or `pause`.

Propagating pause signals across pipelined tasks: The master task of a pipeline-parallel region relays the `continue` or `pause` notification to all other tasks in the region, following the structure of the pipeline. This is achieved by placing *send* instructions to all directly connected tasks in the loop header of the master task (before any other instruction, see `send(st,channels[])` in Figure 4(b)), and matching *receive* instructions (via `get_status()`) in the corresponding location of each of the connected tasks. Subsequent send-receive messages are placed analogously down the stages of the pipeline. This ensures that all parts of an iteration, scattered across pipeline stages, pause appropriately.

All send-receive operations occur over the same point-to-point communication channels used for dependency flows. Upon receiving a `pause` signal, a task explicitly flushes all outgoing channels, transmitting all pending items down the pipeline. This mechanism ensures that all channels are drained properly, relying on the property that at the end of a flushed iteration, all relevant incoming data has been received and processed.

Pausing process: Upon receiving a `pause` signal, a task exits the parallel region by jumping to an exit block which returns `task_paused`. Exiting a parallel region on a pause is identical to exiting the region upon reaching the end of the loop. Indeed, the block returning `task_paused` contains the same instructions as the block returning `task_complete`. These instructions include flushing outgoing channels (see Figure 4(b)). After exiting the iteration, the thread waits on a barrier for other threads executing the parallel region to exit as well (see Algorithm 1).

To resume execution, Morta launches the set of tasks determined to be optimal for the new execution environment. Each task first executes T_{init} when launched (the task initialization function, see Figure 4(b)). Section 5 discusses the overhead of pause and resume as well as the means to alleviate it.

3.4 Hooks for Autonomous Monitoring

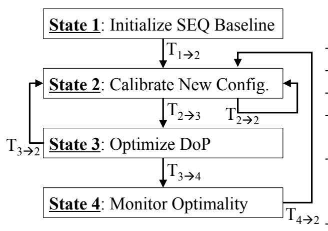
An important aspect of Parcae is task execution time monitoring by Decima. Decima distinguishes between the time a task spends computing and the time it spends waiting for communication, possibly across multiple parallel instances. Morta relies on this information to optimize program configurations. To enable such monitoring, Nona inserts `begin` and `end` hooks into the code of each task. These hooks obtain timestamps using the `rdtsc` instruction on x86 platforms (whose overhead is presented in Section 6.5). Nona inserts `end` immediately before each *send* and *receive* instruction, and `begin` immediately after each *send* and *receive* instruction. Nona also inserts `begin` into the entry block of a task and `end` into the `task_executing` exit block.

The `begin` and `end` hooks help calculate the total compute-time of an instance by accumulating local time intervals (between consecutive {`begin`, `end`} pairs). These local compute-times of each instance then update a global compute-time counter per task, which feeds Decima (described in Section 4), requiring no inter-thread synchronization. Note that the total execute-time of an instance can easily be reported too, as the time elapsed between the first `begin` and the last `end`, from which the communication overhead can be derived. The latter may be important for affinity and allocation optimization purposes; however, these effects were found to be insignificant on our evaluation platforms and experiments.

4. Online Monitoring and Optimization

The goal of the Morta run-time system is to rapidly find parallelism configurations that are optimal for the execution environment. This work focuses on the following optimization objective: minimize total execution time, and subject to that, minimize energy consumption. Morta achieves this objective by maximizing iteration throughput (number of iterations processed per second) and saving idle threads, as explained in this section. The system design allows additional goals, as desired.

A parallelism configuration consists of: (1) a Parallelization Scheme (prepared by Nona, see Section 3), which maps each loop to one of the following: $\mathcal{S} = \{\text{DOANY}, \text{PS-DSWP}, \text{SEQ}\}$; and (2) a Degree of Parallelism (DoP) D, the varying number of threads allocated to every parallel task of DOANY or PS-DSWP schemes. A configuration also contains an assignment of threads to cores, but this aspect was not significant on our evaluation platforms.


State 1: Initialize SEQ Baseline
$T_{1\to2}$
State 2: Calibrate New Config.
$T_{2\to3}$
$T_{2\to2}$
$T_{3\to2}$
State 3: Optimize DoP
$T_{3\to4}$
State 4: Monitor Optimality
$T_{4\to2}$

(a) Finite State Machine Structure

$T_{1\to2}$	Measured SEQ baseline, reconfigure to parallel scheme.
$T_{2\to3}$	Feed configuration profile.
$T_{3\to2}$	Reconfigure to next scheme.
$T_{3\to4}$	Reconfigure to optimal DoP.
$T_{4\to2}$	Detected change in workload, re-calibrate configuration.
$T_{2\to2}$ $T_{3\to2}$ $T_{4\to2}$	Detected change in resource allocation, re-calibrate configuration.

(b) Description of Transitions

Figure 5. Run-time Controller

Morta uses the following schema to identify optimal configurations: establish a baseline performance metric; identify an optimal configuration by repeatedly searching for a better configuration, pausing execution to change configurations, and measuring the performance of the new configuration relative to the baseline; monitor optimality of current configuration and trigger a new search if the dynamic execution environment changes. The finite state machine shown in Figure 5 implements the above schema.

State 1: Initialize Sequential Baseline. When a program enters a parallel region, Morta selects the sequential scheme ($S' =$ SEQ, $D' = \{1\}$) and monitors its execution to establish a baseline throughput T_{seq}. After completing a fixed number of N_{seq} iterations (set to 10 in the current implementation), Morta reconfigures the program to execute in an initial parallel scheme $S' = S_{\text{par}}$ and default DoP $D' = D_{\text{par}}$, and transitions to State 2. (In our current implementation, $S_{\text{par}} =$ PS-DSWP and D_{par} is determined as described later in Section 4.1.) Note that D' is a vector, every element of which represents the DoP of a single task. Also, the initial value of D' need not be 1 (explained in Section 4.1).

State 2: Calibrate New Configuration. In State 2, Morta treats the current parallel configuration with its scheme and DoP (S', D') as being new. It gathers initial timing information for scheme S' while repeatedly reconfiguring the system to $D' \pm 1$ and executing each configuration for a number of iterations N_{par}[3]. This is done to establish a direction for the search for an optimum DoP in the next state. After completing the calibration iterations, Morta restores the original (S', D') configuration and moves to State 3.

State 3: Optimize Degree of Parallelism (DoP). Based on the information collected in State 2, Morta performs a local monotonic search for an optimal DoP for each task, repeatedly reconfiguring the system to different values of DoP and executing a number of N_{par} iterations to measure its performance. Specifically, it uses a finite difference gradient ascent control law [15, 39], as described in detail in Section 4.1. This process converges to an optimal DoP (D'') for the given scheme S', with associated throughput T''. If T'' is better than the best throughput T^* achieved so far for the region, Morta updates T^* with T'' and records S', D''. ($T^* = T_{\text{seq}}$ initially.) Morta then selects the next scheme S' from $\mathcal{S}$, resets DoP to its default $D' = D_{\text{par}}$, reconfigures the system accordingly, and returns to State 2. If all schemes in $\mathcal{S}$ have been explored, Morta retrieves the configuration S', D'' that achieved the best throughput T^*, reconfigures the system accordingly, and moves to State 4.

State 4: Monitor Configuration Optimality. In this state, the Decima monitor performs a passive monitoring of the system (i.e., no reconfigurations) to detect changes in either the resources allocated to the program or in the workload of the program itself. For the former, Decima interacts with the platform-wide run-time system (described in Section 4.2). For the latter, Decima monitors the throughput of the parallel region; if the throughput changes by more than a preset threshold, the workload is deemed to have changed. If any such change is detected, the configuration is suspected to have become suboptimal, and control returns to State 2 retaining the current scheme. If the change corresponds to an increase in resources, the current DoP is retained (hopefully as a good starting point); otherwise, if resources decreased or the throughput of the workload itself decreased, the DoP is reset to its initial value D_{par}.

Note that all reconfigurations that modify the DoP while keeping the scheme intact (such as those carried out by State 2, State 3, and potentially upon transitions $T_{3\to4}$ and $T_{4\to2}$) do not require changing the code distributed among the worker threads. Reconfigurations that do modify the parallelization scheme, upon transitions $T_{1\to2}$ and $T_{3\to2}$, do involve replacing this code.

[3] The number of iterations N_{par} is dynamically set to $\max(N_{\text{seq}}, 2 \cdot d_P)$, where d_P is the current DoP of the parallel task being optimized.

Algorithm 2: Optimizing multiple DoPs in a region

```
Input: Calibrated parallel region, thread budget $N$
Output: Optimized DoP for region ($\leq N$)
totalDoP ← computeTotalDoP(region.parallelTasks())
foreach $P_i \in$ region.parallelTasks() do
    $P_i$.opt ← false; $P_i$.sat ← false
repeat
    optimize_a_task ← false
    $\mathcal{P}$ ← sortInAscendingThroughput(region.parallelTasks())
    for $P_i \in \mathcal{P}$ while ¬optimize_a_task do
        $\overline{d_{P_i}} \leftarrow N - \text{totalDoP} + d_{P_i}$
        if $(\neg P_i.\text{opt}) \vee ((d_{P_i} < \overline{d_{P_i}}) \wedge (\neg P_i.\text{sat}))$ then
            $d_{P_i}$ ← gradientAscent($P_i, d_{P_i}, \overline{d_{P_i}}$)
            $P_i$.opt ← true; $P_i$.sat ← $(d_{P_i} < \overline{d_{P_i}})$
            totalDoP ← updateTotalDoP(totalDoP, $d_{P_i}$)
            optimize_a_task ← true
until ¬optimize_a_task
if parThroughput(region) > (0.9*totalDoP)*seqThroughput(region) then
    return totalDoP        // current parallel scheme profitable, keep it
else if bumpToNextScheme(region) then
    return calibrateAndOptimize(region) // try next scheme recursively
else return 1 // no parallel scheme is profitable, revert to sequential
```

4.1 Optimizing the Degrees of Parallelism

A given parallelization scheme may comprise both sequential $\{S_1, S_2, \ldots, S_m\}$ and parallel tasks $\{P_1, P_2, \ldots, P_n\}$. The degree of parallelism d_{S_i} for every sequential task is inherently 1. The objective of optimizing DoP (State 3 of Figure 5(a)) is therefore to find a degree of parallelism d_{P_i} for each parallel task that maximizes the overall performance of the region. The problem can be formulated as follows:

Maximize overall throughput: $T = f(d_{P_1}, d_{P_2}, \ldots, d_{P_n})$

subject to: $d_{P_i} \geq 1 \;\; \forall i, \quad \sum_{i=1}^{n} d_{P_i} \leq N$

where N denotes the total number of threads available to the program minus those used by its sequential tasks (m). Morta optimizes each d_{P_i} separately, in turn, according to the relative throughputs of the tasks in ascending order (see Algorithm 2). This is done in order to prioritize slower tasks, which are typically bottlenecks in pipelined networks of sequential-parallel tasks.

Morta computes the optimal DoP d_{P_i} for a parallel task P_i by using a fast iterative gradient ascent technique [39], based on the assumption that function f is unimodal with a single (local) optimum. Section 6.4 discusses the impact of this assumption. On each iteration of the optimization routine, the current throughput is compared with the previous throughput[4], and this difference either establishes the gradient of the change to the next DoP or concludes the search. First, an upper bound $\overline{d_{P_i}}$ on the allowed DoP is calculated, which is equal to the maximum number of threads currently available for P_i. Initially, every parallel task is assigned half of its fair share of threads: $\frac{N}{2n}$. Thus, $\overline{d_{P_i}} = N - \sum_{j \neq i} \frac{N}{2n} = \frac{(n+1)N}{2n}$. The search for an optimal $d_{P_i} \in [1, 2, \ldots, \overline{d_{P_i}}]$ starts at the midpoint of this range: $d_{P_i}(0) = D_{\text{par}} = \frac{1}{2}\overline{d_{P_i}}$. Morta then sets $d_{P_i}(1) = d_{P_i}(0) \pm 1$ according to whichever achieves greater throughput. We call the search *increasing* or *decreasing*, respectively.

[4] Throughput is measured directly as the number of iterations executed in a certain amount of time, rather than computed analytically by relying on some model of the throughput function f.

Next, the gradient between the current $(k+1)$ solution and the previous (k) solution is calculated, starting from $k = 0$:

$$\delta(k+1) = T(d_{P_i}(k+1)) - T(d_{P_i}(k)). \quad (1)$$

If $\delta(k+1) < 0$, the optimal solution has been passed and Morta terminates the search taking $d_{P_i}(k)$ to be the solution. Recall that we seek to maximize the throughput, and subject to that, minimize the number of threads (thereby saving energy). The case $\delta(k+1) = 0$ is treated similar to case $\delta(k+1) < 0$ above if the search is increasing, and similar to case $\delta(k+1) > 0$ below if the search is decreasing. If $\delta(k+1) > 0$, the next solution is calculated according to the gradient ascent formula:

$$d_{P_i}(k+2) = d_{P_i}(k+1) + \alpha\delta(k+1) \quad (2)$$

where α is positive if the search is increasing, and negative if the search is decreasing. The search continues by incrementing k and evaluating Equation 1 and Equation 2, repeatedly.

After optimizing d_{P_i}, the process repeats by sorting the tasks according to their throughputs, and selecting the next task P_j to optimize. The process terminates after all d_{P_i}'s have been optimized, possibly more than once, and cannot be further improved.

Once the search terminates at an optimal DoP, Morta measures the overall throughput of the region $T = f(d_{P_1}, \ldots, d_{P_n})$ and compares it with the baseline sequential throughput T_{seq} measured in State 1. The parallel configuration is deemed profitable only if its efficiency is significantly better than that of the sequential configuration. Otherwise, Morta repeats the optimization process considering an alternative parallel scheme exposed by Nona, if any (corresponding to transition $T_{3\to2}$ in Figure 5). When all available schemes have been considered, Morta chooses the most efficient scheme (possibly even SEQ) for execution, and enters State 4. To accelerate the optimization process, Morta caches previously optimized configurations and reuses them, if feasible, as initial configurations upon future entry into a parallel region.

4.2 Platform-wide Control

The description of the Morta run-time system in Section 4.1 applies to a single program executing a parallel region. However, the same control system used to optimize the parallel execution of one program easily extends and generalizes to multiple programs running concurrently, thereby achieving platform-wide execution optimization. Indeed, each executing program p can be considered as a collection of parallel and sequential tasks, executed in parallel and independent of other co-scheduled programs, as if all programs belong to one parallel region with multiple parallel tasks. At this outer level, Morta must decide how to partition the total number of threads $\mathcal{N}$ available in the system across the different co-scheduled programs: $\mathcal{N} = \sum_p N_p$, such that a specified optimization goal is met. Given its budget of threads N_p, a program is optimized and monitored by its dedicated controller. Algorithm 3 summarizes the platform-wide control logic to minimize the overall execution time of a batch of co-scheduled programs.

The platform-wide Morta run-time system is implemented as a daemon, launched upon system boot. When a flexible parallel program p is launched, its controller registers itself with the daemon and acquires its fair share of resources N_p; initially $N_p = \mathcal{N}/\mathcal{P}$ where $\mathcal{P}$ is the number of flexible parallel programs. Each controller proceeds to initialize, optimize, and monitor its program, as explained (through States 1,2,3 and 4), and reports its optimal amount of resources $N'_p \le N_p$ to the daemon (upon transition $T_{3\to4}$).

On receiving optimization results N'_p from the controllers, the daemon distributes slack resources $\mathcal{N} - \sum_p N'_p$ if any, among controllers with $N'_p = N_p$. Finally, the daemon monitors changes

Algorithm 3: Platform-wide Optimization

```
Input: platform-wide thread budget $\mathcal{N}$
Output: Optimized DoP for platform ($\le \mathcal{N}$)
slack ← $\mathcal{N}$
activePrograms ← platform.programs
while slack > 0 do
    foreach $P \in$ activePrograms do
        $N_{P_{slack}}$ ← slack/activePrograms.size
        slack ← slack - $N_{P_{slack}}$
        $N_P \leftarrow N_{P_{new}} + N_{P_{slack}}$    // $N_{P_{new}} \leftarrow 0$ initially
        $N_{P_{new}}$ ← P.optimizeDoP(P.region, $N_P$)
        if $N_{P_{new}} < N_P$ then
            slack ← slack + $(N_P - N_{P_{new}})$
            activePrograms.extract(P)
```

in system resources (launch and termination of programs) and repartitions resources across executing programs when they occur.

5. Reducing Run-time Overheads

The Morta run-time system's overheads include:

1. Task Migratability: consisting of (i) Task Activation, yielding to and returning from the task activation loop; and (ii) Data Management, loading and saving cross-iteration dependency data; both occurring per iteration.
2. Pause-Resume: on receiving a pause signal, all threads synchronize by means of a barrier; threads which reach the barrier early waste cycles waiting for the slower threads to catch up (shown as Barrier Wait in Figure 2). Tasks must be re-started on threads on resumption.
3. Parallelism Reconfiguration: time spent executing core optimization routine to determine new parallelism configurations (shown as Parallelism Reconfiguration in Figure 2).
4. Status Query: querying Morta whether the program should pause, after each iteration.
5. Monitoring: recording timestamps for execution time statistics.

On our evaluation platforms, the last two were barely noticeable. The remaining overheads (measured and reported in Section 6.5) can be significantly reduced in the common case:

1. Task Migratability: Rather than returning to the task activation loop every iteration as in Figure 4, an optimized (yet non-migratable) version may retain the original loop backedge. This makes it possible to hoist the load out of the loop header into the preheader, and to push the corresponding store into the pause exit block (see Figure 4). Note that this only constrains the migration of sequential tasks in between parallelism reconfigurations since sequential tasks have dependencies flowing across task instances.
2. Parallelism Reconfiguration: The thread executing the master task is the first to receive the pause signal, directly from Morta. This thread can immediately start executing the optimization routine to determine the next parallelism configuration, rather than waiting for other threads. This overlaps the time spent executing the optimization routine with the gap between the last and first threads to reach the barrier.
3. Pause-Resume: During gradient ascent, which performs most reconfigurations, the degree of parallelism (DoP) of a parallel task is increased or decreased. Recall that a parallel task is explicitly constructed such that it either does not have any loop-carried dependencies or updates global memory with appropriate synchronization. Consequently, additional threads may start executing the parallel task without violating program consistency, eliminating the need for a full barrier wait. In the pres-

ence of communication with other tasks, however, actions appropriate to the communication policy must be taken. As described in Section 3, a sequential task sends (receives) dependencies in round-robin order to (from) each of the threads executing the parallel task. The sequential task must process all tokens up to the iteration at which the parallel region was paused, before changing the width of the communication channel between itself and the parallel task whose DoP is being optimized. The optimized Morta run-time system can ensure this by leveraging the iteration counts of each task.

6. Evaluation

Table 2 describes the two **real** platforms used to evaluate Parcae. The Nona compiler is built on LLVM [19] revision 129859. Nona generates optimized assembly which is then assembled and linked by GCC version 4.4.1. Parcae is compared with an LLVM compiler also built on revision 129859, applying fixed high-quality parallelization schemes (PS-DSWP [34, 43] and DOANY [31, 46]) targeting an unloaded parallel platform, and using a standard threadpool with load balancing by the Linux scheduler [31, 43]. The Morta run-time system is implemented on top of the same threadpool, facilitating fair comparison.

As a second point of comparison, the optimal program configuration is determined for each workload by enumerating all possible static parallelism configurations of the program and picking the best performing one. The performance of Parcae is then compared against that of the optimal configuration. The execution time of the sequential program produced by the same set of optimizations excluding the parallelization transformations serves as the baseline for speedup computation of all parallel runs. All execution times used to compute speedups in Table 1 are averages over N runs, where $N \geq 3$ is the minimum number of runs required for a standard deviation of less than 5% of the geomean execution time.

Processors will have interfaces to turn cores on or off at fine time granularities [8]. To evaluate energy savings that Parcae could deliver by leveraging such capabilities, we measured the power consumption characteristic of each platform by executing a power-virus that maintains 100% CPU utilization of each core to measure power draw when a number of cores are busy, and by leaving the system unused to measure power draw when those cores are idle. The processor's dynamic power range is then the difference between active and idle power. Active processor power consumption as a fraction of whole platform power consumption is 18.0% for Platform 1 and 26.7% for Platform 2. Processor energy is then computed as the integral of the piece-wise product of time spent actively using a number of cores and the average power draw of those many active cores. Recent work uses a similar methodology [8].

Table 1 describes the benchmarks selected to evaluate Parcae. These include benchmarks parallelized by the baseline state-of-the-art compiler [31, 43], allowing direct comparison. The last three columns of the table provide a summary of the execution time and energy improvements delivered by Parcae. These improvements are discussed in detail below. In the interest of space, we present case studies using one application for three interesting scenarios, each highlighting a different aspect of Parcae.

6.1 Parcae Adapts Execution to Workload Change

`blackscholes` calculates the prices for a portfolio of European options using the Black-Scholes partial differential equation [7]. To study workload variation, the source code is modified to enable pricing of two different portfolios. Nona is able to apply the PS-DSWP parallelization scheme, in addition to SEQ. Figure 6(b) shows Morta and Decima in action. Morta starts in State 1 to initialize the sequential baseline. It then enters State 2 to set the initial configuration and determine the direction for gradient ascent.

Feature	Platform 1	Platform 2
Brand Name	Intel Xeon® E5310	Intel Xeon® X7460
Processor	Intel Core®, 64-bit	Intel Core®, 64-bit
Clock Speed	1.60 GHz	2.66 GHz
Total # Cores	8	24
Total L2 Cache	16 MB	36 MB
Total L3 Cache	-	64 MB
Total RAM	8 GB	24 GB
OS	Linux 2.6.32	Linux 2.6.31

Table 2. Hardware Platforms Used for Evaluation

In State 3, it performs gradient ascent until peak throughput is achieved, at which point ($t = 5.0s$) it enters State 4 to monitor the execution of this configuration. During the optimization phase, the re-configuration interval constantly varies according to the optimization logic; this results in compute throughput measurements that are accurate relative to each other, but are less accurate with respect to the baseline throughput. Hence, once the peak is reached, the system stabilizes for additional iterations to obtain a more accurate throughput measurement. Decima's workload change detection logic kicks in after this throughput measurement. At $t = 56.6s$, Decima detects workload change via drop in throughput, and signals Morta to re-optimize in response. In State 3, Morta reduces DoP by performing decreasing gradient ascent, finally to reach State 4 at $t = 79.9s$; Decima monitors the remaining execution. Parcae achieves performance within 1.4% of optimal (due to overheads) while consuming 45.4% less energy.

6.2 Parcae Optimizes Across Multiple Parallelization Schemes

`geti` is a data mining program that computes a list of frequent itemsets using a vertical database. The set semantics of the output operations enables order relaxation via commutativity, which is expressed using 11 annotations in 889 lines of source code [31]. The resulting dependence graph is then amenable to PS-DSWP and DOANY parallelization schemes (in addition to SEQ). Figure 6(c) shows the execution trace. As with `blackscholes`, Morta starts in State 1 to initialize the sequential baseline. In State 2, it calibrates an initial configuration of the PS-DSWP scheme, and starts an increasing gradient ascent in State 3. Morta determines an optimal configuration for the PS-DSWP scheme (at $t = 14.0s$), and compares throughput with that of the SEQ scheme. As the comparison is below the preset threshold, Morta returns to State 2 in order to evaluate the next parallel scheme: DOANY. Optimizing the DoP for DOANY then takes place in State 3 (from $t = 16.8s$ until $t = 26.0s$). The optimal DOANY configuration is found to be of adequate throughput, so Morta moves to State 4 and Decima monitors the remaining execution. Parcae improves performance over the baseline by 12.6%, and achieves performance within 18.6% of optimal. Parcae reduces energy consumption by 23.9%.

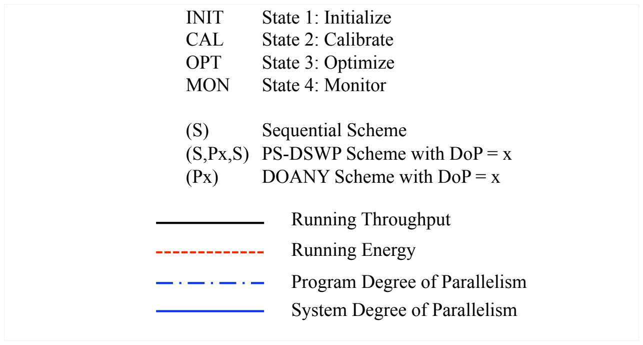
6.3 Parcae Adapts Execution to Resource Availability Change

Figures 6(d) and 6(e) illustrate Morta's platform-wide control capability, with the former highlighting control of programs with homogeneous characteristics while the latter heterogeneous. In both cases, the controller primarily aims to minimize the overall execution time of the batch of programs. Simultaneously, by virtue of fair allocation of slack resources to all active programs on each round of optimization in Algorithm 3, the platform-wide controller also seeks to avoid adversely impacting any one program.

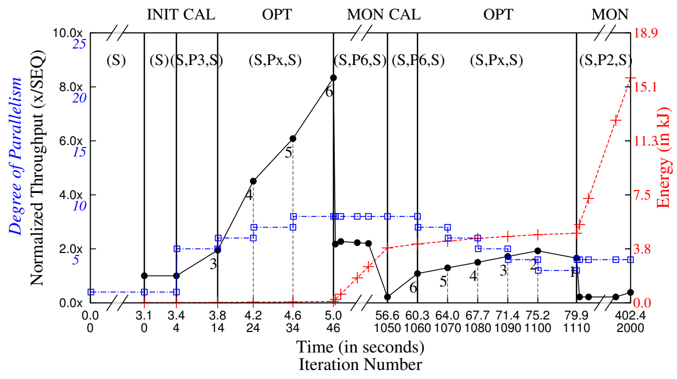
Homogeneous characteristics: Eight copies of `blackscholes` processing the same input are launched in succession on Platform 2 having 24 cores. For brevity, the execution of only one copy P_0 is shown (labeled *Program* P_0). The figure also shows platform-wide

Program	Origin	Main Loop	Parallel Coverage	Total number of		Parcae Improvement relative to		
				Invocations	Iterations	Baseline [31, 43]		Optimal
						Exec. Time	Energy	Exec. Time
blackscholes	Seq. version from PARSEC [7]	`worker`	92.94%	1	1000	-1.39%	45.35%	-1.39%
geti	MineBench [26]	`FindSomeETIs`	99.99%	24	12242	12.55%	23.90%	-18.56%
kmeans	STAMP [25]	`work`	99.30%	501	131334144	41.94%	83.89%	-62.98%
ks	LLVM Test Suite [21]	`FindMaxGpAndSwap`	99.97%	7750	488250	3.78%	35.11%	-0.83%
md5sum	Open Source [4]	`main`	99.99%	1	384	34.41%	58.51%	-3.55%
potrace	Open Source [38]	`main`	100.00%	1	200	9.95%	42.67%	-2.97%
url	NetBench [24]	`main`	100.00%	1	40000	-0.18%	36.22%	-2.40%

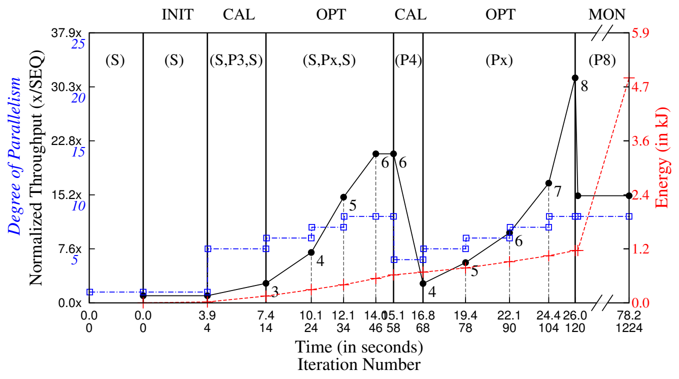
Table 1. Sequential programs transformed, their origin, transformed loop, fraction of total execution time spent in parallelized loop, number of invocations and iterations of the parallelized loop, and improvements in execution time and energy delivered by Parcae. The improvements are on a single input on Platform 2, and are relative to baseline parallel and optimal parallel versions of these programs. Optimal versions are determined via offline exhaustive search. Positive numbers indicate improvement.



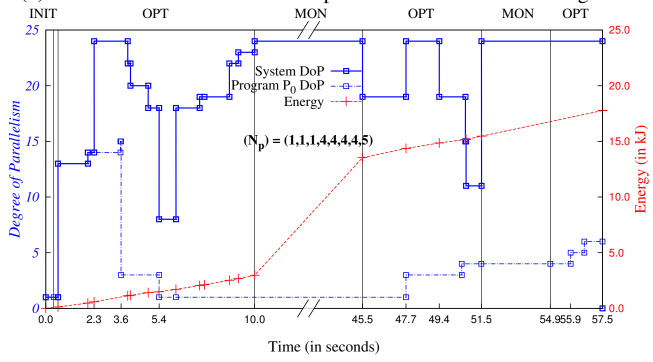
(a) Legend



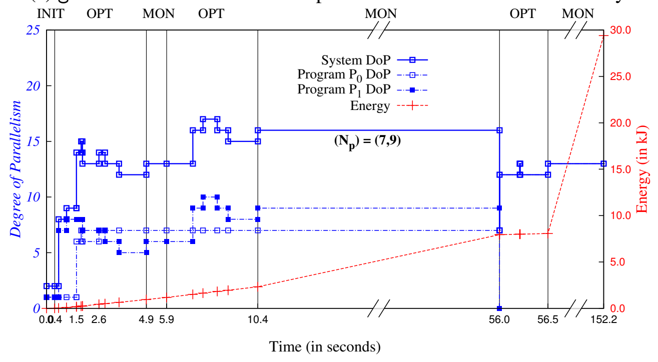
(b) `blackscholes`: Controller re-optimizes when workload changes



(c) `geti`: Controller determines optimal execution scheme from many



(d) Controller optimizes eight homogeneous programs simultaneously



(e) Controller optimizes two heterogeneous programs simultaneously

Figure 6. Parcae run-time control. Solid vertical lines indicate state transition times. All throughput values are normalized to the throughput measured in the INIT state. In (b) and (c), the state transitions shown at the top of each figure are those of an individual program's controller, whereas in (d) and (e), they are the transitions of the platform-wide Morta daemon.

dynamic thread utilization as determined by the platform-wide daemon (labeled *System*). P_0 begins execution on an unloaded machine, so following the initialization of State 1, State 2 calibrates the parallel task of the PS-DSWP scheme starting from a DoP of $d_{P_0} = \frac{1}{2}\overline{d_{P_0}} = 11$. (There are two additional sequential tasks resulting in a whole program DoP of 13.) P_0's controller then transitions to State 3 and performs an increasing gradient ascent. By $t = 3.6s$, all eight program copies have launched, and the daemon reallocates threads across them: each copy is assigned a DoP of 3. In response, P_0's controller transitions to State 2 and measures the parallel performance of configuration (S,P1,S), which is now the only feasible parallel configuration. It then computes the configuration's efficiency and compares the efficiency with that of sequential execution, determining at time $t = 5.4s$ that sequential execution is preferred, and returning 2 threads back to the daemon. Spare threads are redistributed by the daemon among the copies repeatedly. By $t = 10.0s$, the platform-wide optimization algorithm converges (at $\{N_p\} = \{1, 1, 1, 4, 4, 4, 4, 5\}$), all copies enter their stable configurations, and the controllers enter State 4 (the monitoring state). At $t = 45.5s$, the fastest copy terminates, yielding its (5) threads back to the daemon, which in turn initiates another round of optimization. The figure shows how, barring the short optimization phases, Parcae enables full use of all threads on the platform. Compared to OS scheduling of the eight copies of the baseline 24-thread parallel version, Parcae reduces total execution time of the workload by 12.8%, and energy consumption by 29.1%.

Interestingly, the benefit in execution time using Parcae increases as the number of concurrently executing programs increases. With sixteen copies, execution time and energy improve by 18.3% and 52.7%, respectively. With twenty four copies, improvements reach 22.4% and 22.4%, respectively. The co-operative release and acquisition of resources by Parcae reduces contention among concurrently executing programs. Scheduling and memory contention significantly overwhelm the benefits of parallel execution when copies of the baseline parallelized programs execute each with 24 threads.

Heterogeneous characteristics: One copy each of kmeans (labeled *Program* P_0) and blackscholes (labeled *Program* P_1) are launched in succession on Platform 2 having 24 cores. kmeans is a DOANY parallel loop with a single parallel task whereas blackscholes is a PS-DSWP parallel loop with two sequential tasks and one parallel task. Additionally, the two programs differ in parallelization scalability, average time per iteration, and total execution time. The platform-wide daemon initiates a round of optimization at $t = 0.4s$ when both programs have launched, and assigns each program its fair share DoP of 12. It then optimizes each program in turn starting with P_0. In response, P_0's optimizer performs gradient ascent to determine optimal use of the assigned threads. At $t = 2.6s$, P_0 determines that (P7) is the optimal parallelism configuration and returns 5 threads back to the daemon. In the meantime, P_1's controller performs gradient ascent to determine that the optimal parallelism configuration is (S,P4,S) and returns 6 threads back to the daemon at $t = 4.9s$. The daemon then initiates another round of optimization at $t = 5.9s$ by assigning the slack threads to P_1 via a DoP assignment of 17. P_1's controller performs gradient ascent again to settle at (S,P7,S). By $t = 10.4s$, the platform-wide optimization algorithm has converged (at $\{N_p\} = \{7, 9\}$), and the individual controllers of both programs are in the monitoring state. At $t = 56.0s$, program P_1 finishes and yields its threads back to the daemon, which in turn initiates another round of optimization by allocating those threads to P_0. P_0's controller performs gradient ascent and determines that (P13) is the ideal parallelism configuration and enters the monitoring state. It stays in that state until P_0 finishes at $t = 152.2s$.

Operation	**Overhead**		
	Platform 1	Platform 2	Unit
Task Migratability			
- Task Activation	11	7	nanoseconds per task instance
- Data Management	3	2	nanoseconds per dependency per task instance
Status Query	8	5	nanoseconds per query
Monitoring	44	15	nanoseconds per timestamp
Pause-Resume	33	4	milliseconds per pause-resume
Parallelism Reconfiguration	35	2	milliseconds per reconfiguration

Table 3. Recurring run-time overheads

Compared to OS scheduling of the baseline 24-thread parallel versions of the two benchmarks, Parcae reduces total execution time of the workload by 53.5%, and energy consumption by 76.3% by virtue of reducing the total execution time and using fewer cores. Parcae improves the execution time of each benchmark as well: kmeans improves by 53.5% and blackscholes by 36.7%. Recall that on a platform with no contention from other programs, the Parcae version of blackscholes took longer to execute than the baseline version (Table 1). Parcae's benefits manifest when the same program is executed on the same platform but with contention from other programs.

6.4 Optimality: A Closer Look

While Parcae delivers significant performance improvements, its achievements are still short of optimal execution determined via offline exhaustive search (see Table 1) for two main reasons. For blackscholes, geti, md5sum, and potrace, Parcae identifies the optimal configuration, but converges to that solution after first spending multiple iterations in SEQ and other sub-optimal schemes. In addition to the above overhead, for kmeans, ks, and url, the assumption about unimodal performance characteristic with increasing DoP (discussed in Section 4.1) does not hold; Parcae identifies a local optimum, which is worse than the global optimum. The Parcae gradient ascent algorithm can be enhanced to escape from local optima. We leave this to future work.

6.5 Morta and Decima Overheads

The execution time improvements reported in Table 1 account for all overheads of the system. Table 3 shows each overhead independently. The first three overheads are recurring (incurred on each instance of each task). To quantify them, we executed microbenchmarks designed to exercise each overhead independently. All aspects of the run-time control system are implemented in shared memory, thus making the run-time operations extremely lightweight. In the common case, task instances are executed on the same core, resulting in low-latency access to cross-iteration dependency data. Timestamp acquisitions are not serializing instructions; their latencies may be masked naturally via hardware exploiting instruction-level parallelism. By comparison to the geomean time per iteration across all benchmarks (1552.30 microseconds), the recurring overheads incurred on each iteration (or task instance) are orders of magnitude lower. "Pause-Resume" and "Parallelism Reconfiguration" overheads are application-specific and DoP-specific. The overheads shown in these two categories are for blackscholes with maximum DoP.

7. Related Work

Auto-parallelization: Most work on auto-parallelization at compile-time (e.g., [34, 43, 47]) or run-time (e.g., [18, 30, 35, 37]) is primarily concerned with parallelism discovery and extraction. Parcae addresses the complementary problem of parallelism tuning and optimization. Parcae tunes parallelism dynamically, and is designed to work with multiple automatic parallelization passes.

Flexible Parallel Execution: Adaptive Thread Management throttles the number of threads allocated to DOALL loops during execution, as their measured speedup exceeds or falls short of an expected speedup [13, 14]. Dynamic feedback has been used to determine the best synchronization policy from among those exposed by the compiler [12]. The ADAPT optimizer applies loop optimizations at run-time to create new variants of code [44]. Parcae employs multiple parallelization schemes prepared at compile-time to support efficient optimization at run-time, and optimizes multiple programs co-scheduled on a system. A key novelty over prior work is the mechanism for efficient pausing of one set of communicating tasks, followed by the resumption of the parallel region's execution by a different set of communicating tasks.

Auto-tuning: Auto-tuning is used to optimize parameters that impact performance. The optimal number of threads in an application's thread-pool can be determined across multiple application runs [16, 45]. For linear algebra codes, optimal loop tiling factors can be determined at compile-time for each platform [3, 5, 32]. However, not all important architectural variables can be handled by such parameterized adaptation; some choices require changing the underlying source code. This type of adaptation involves generating distinct implementations for the same operation [3]. The PetaBricks [2] and Elastin [27] frameworks leverage alternative code implementations exposed by programmers. Parcae automatically generates multiple versions for each parallel region from standard source code, and selects and tunes them at run-time based on performance feedback.

Parallelization Libraries: Numerous interfaces and enabling runtime systems have been proposed to adapt parallel program execution to run-time variability [10, 11, 22, 33, 40]. These systems burden the programmer with the task of extracting parallelism and with the task of expressing parallelism such that it can be optimized. Parcae minimizes the burden on the programmer as far as parallelism extraction is concerned, and fully automates the task of rewriting the parallel program for flexible execution. The runtime systems in those works either require the programmer to specify the degree of parallelism thus limiting performance portability (e.g. Cilk++ [20]), or are not adaptive (e.g. the TBB Auto Partitioner [36]), or handle only counted DOALL parallel loops (e.g. the Lazy Splitting scheduler [42]), or only optimize a single program in isolation; Parcae optimizes multiple programs running concurrently, handles counted and uncounted loops with different forms of parallelism, adapts parallelism configurations to changes in the environment, and delivers portable performance.

Operating Systems: Future operating systems for multicores will need tighter integration between parallel application run-time systems and operating system schedulers [6, 9, 28]. Parcae demonstrates the benefits of such an integration. Parcae's monitoring and resource management services could be integrated with operating systems for manycore processors such as Barrelfish [6].

8. Conclusion

This paper presented Parcae, a system that automatically generates flexible parallel programs and that monitors and optimizes the execution of multiple flexible programs running on a shared parallel platform. Compared to conventional parallel execution of seven benchmarks, Parcae reduced execution time by -1.4% (a small slowdown) to 41.9% with concomitant estimated processor energy savings of 23.9% to 83.9%. While Parcae was evaluated in the context of two parallelizing transforms, it can be extended to support additional, new ones. Parcae can improve the performance of flexible parallel programs even when co-scheduled with conventional, inflexible programs. With promising advances in auto-parallelization and increasing diversity in execution environments, a holistic, automatic, dynamic, and generally applicable parallelism tuning system will only become more relevant.

Acknowledgments

We thank the members of the Liberty Research Group for their support and feedback. We thank Carole-Jean Wu and Lennart Beringer for feedback on an early draft of this paper. We thank the anonymous reviewers for their insightful comments. This material is based on work supported by National Science Foundation Grant 1047879. Jae W. Lee was partly supported by the Korean IT R&D program of MKE/KEIT KI001810041244. This work was carried out while Arun Raman was a graduate student at Princeton University, and while Ayal Zaks was visiting Princeton University, supported by the HiPEAC network of excellence, and on leave from IBM Haifa Research Lab.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[3] C. W. Antoine, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:2001, 2000.

[4] Apple Open Source. md5sum: Message Digest 5 computation. http://www.opensource.apple.com/darwinsource.

[5] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 219–228, 2009.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[8] O. Bilgir, M. Martonosi, and Q. Wu. Exploring the potential of CMP core count management on data center energy savings. In *Proceedings of the 3rd Workshop on Energy Efficient Design (WEED)*, 2011.

[9] S. L. Bird and B. J. Smith. PACORA: Performance aware convex optimization for resource allocation. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar: Posters)*, 2011.

[10] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Computing*, 33(10-11):700–719, 2007.

[11] Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. Adapting application execution in CMPs using helper threads. *Journal of Parallel and Distributed Computing*, 69(9):790 – 806, 2009.

[12] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the 18th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1997.

[13] G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz. Compiler and runtime support for programming in adaptive parallel environments. In *Scientific Programming*, pages 215–227, 1995.

[14] M. W. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. In *Proceedings of the 2nd SUIF Compiler Workshop*, 1997.

[15] J. L. Hellerstein, V. Morrison, and E. Eilebrecht. Applying control theory in the real world: Experience with building a controller for the .NET thread pool. *Performance Evaluation Review*, 37:38–42, 2010.

[16] T. Karcher and V. Pankratius. Run-time automatic performance tuning for multicore applications. In *Proceedings of the International Euro-Par Conference on Parallel Processing (Euro-Par)*, pages 3–14, 2011.

[17] A. Kejariwal, A. Nicolau, A. V. Veidenbaum, U. Banerjee, and C. D. Polychronopoulos. Efficient scheduling of nested parallel loops on multi-core systems. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP)*, pages 74–83, 2009.

[18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 211–222, 2007.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[20] C. E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC)*, pages 522–527, 2009.

[21] LLVM Test Suite Guide. http://llvm.org/docs/TestingGuide.html.

[22] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55, 2009.

[23] J. Mars, N. Vachharajani, M. L. Soffa, and R. Hundt. Contention aware execution: Online contention detection and response. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, Toronto, Canada, 2010.

[24] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A benchmarking suite for network processors. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2001.

[25] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[26] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. 2006.

[27] I. Neamtiu. Elastic executions from inelastic programs. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2011.

[28] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 376–387, 2010.

[29] D. A. Penry. Multicore diversity: A software developer's nightmare. *ACM SIGOPS Operating Systems Review*, 43:100–101, 2009.

[30] C. D. Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. In *Proceedings of the 5th International Conference on Supercomputing (ICS)*, pages 252–263, 1991.

[31] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[32] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[33] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using DoPE: the degree of parallelism executive. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[34] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2008.

[35] L. Rauchwerger, N. M. Amato, and D. A. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming (IJPP)*, 26:537–576, 1995.

[36] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2008.

[37] J. Saltz, R. Mirchandaney, and R. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40, 1991.

[38] P. Selinger. potrace: Transforming bitmaps into vector graphics. http://potrace.sourceforge.net.

[39] J. C. Spall. *Introduction to Stochastic Search and Optimization*. Wiley-Interscience, 2003.

[40] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 147–156, 2010.

[41] A. Tiwari and J. K. Hollingsworth. Online adaptive code generation and tuning. In *Proceedings of the 25th International Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[42] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 179–190, 2010.

[43] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 389–400, 2010.

[44] M. J. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *Proceedings of the 1999 International Conference on Parallel Processing (ICPP)*, pages 163–170, 1999.

[45] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 75–84, 2009.

[46] M. Wolfe. DOANY: Not just another parallel loop. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1992.

[47] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.