# From Sequential Programming to Flexible Parallel Execution

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

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

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

## ABSTRACT

The embedded computing landscape is being transformed by three trends: growing demand for greater functionality and enriched user experience, increasing diversity and parallelism in the processing substrate, and an accelerating push for ever-greater energy efficiency. For programmers, these trends give rise to three challenges: writing code for a potentially heterogeneous architecture, extracting parallelism in software, and maximizing a multivariate (performance, power, energy, etc.) fitness function of user satisfaction which may vary with time. To meet these challenges, clarion calls have been issued for programmers to start writing software in new *parallel* programming models. Fundamentally, however, these proposals detract programmers from delivering new features and enriched user experience in the shortest time possible. This paper proposes to attract embedded systems programmers to a vertically integrated approach, comprising extensions to the sequential programming model, a parallelizing compiler, and an optimizing run-time system, to enable them to tackle all three challenges.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Run-time environments*

## General Terms

Design, Performance

## Keywords

adaptivity, code generation, compiler, embedded, flexible, GPGPU, heterogeneous, multicore, optimization, parallel, parallelization, performance portability, run-time, tuning

## 1. INTRODUCTION

In mainstream computing, a heterogeneous multicore architecture comprising a mixture of multiple, different, types of computational elements is becoming the predominant processor design. Heterogeneity is *de rigueur* in embedded systems, but there is also a definite push towards consolidating multicore CPUs, multicore GPUs, and various accelerators onto a single chip. The primary obstacle to effective use of these ostensibly powerful computational substrates is the difficulty in programming them. Two predominant schools of thought exist to overcome this obstacle.

The first school exhorts programmers to write code using new parallel programming models or libraries [1, 5, 6, 11]. These, however, burden the programmer with the need to reason about complex thread interleavings and concurrency control mechanisms, or serialize execution between different computational elements. Often, these models force the programmer to specify a fixed parallelization strategy (e.g. data parallel, pipeline parallel), fixed split of functionality across heterogeneous computational elements (e.g. a CPU program with GPU "kernels"), and fixed concurrency control strategy (e.g. locking, transactions) [7]. This *early binding* of function to form breaks the abstraction between software and hardware, resulting in suboptimal performance when workload or resource availability changes [3, 9].

The second school promises to unburden the programmer of the onerous task of (re-)writing parallel programs, by having the compiler automatically extract parallelism from the program [4, 10]. However, in practice, the widespread use of sequential programming to implement algorithmic specifications imposes severe constraints on the compiler's ability to extract scalable parallelism. As with manual parallel programming, most compilers perform early binding, leading to suboptimal parallel execution in a variety of scenarios [9].

To overcome challenges evident in both approaches, this paper argues for a three-pronged approach consisting of:

- Extensions to the sequential programming model, which relax the constraints on instruction ordering by enabling programmers to specify commutativity relationships between sets of instructions and weaker data consistency requirements. By using these extensions, the programmer does not specify parallelism, but rather *enables* automatic parallelization. Additionally, the programmer embeds hints in code blocks indicating the natural affinity of that code block to a computational unit (to a particular accelerator, for example). However, the system may choose to ignore the hints.

- A parallelizing compiler, which uses the relaxations and hints specified by the programmer to extract multiple types of parallelism from the sequential program,


*CASES'12,* October 7–12, 2012, Tampere, Finland.

generate multiple code versions to target a heterogeneous architecture, and encode the program configuration space induced by parallelization and code versioning in a small set of tunable parameters. Programs created thus are called *flexible parallel programs*.

- An optimizing run-time system, which monitors program performance and system events, such as launches of new programs and resource availability change, and determines the best configurations (settings of the tunable parameters exposed by the compiler, and the dynamic mapping of code to computational element) of all concurrently executing flexible parallel programs. The optimization objective is specified by the user, may vary with time, and may be composed of latency and throughput requirements with constraints on power, memory, energy, temperature, etc.

The remainder of this paper describes synergistic advances in the above three areas, which together may restore the embedded system developer's focus to delivering exciting new features and applications.

## 2. PROPOSED APPROACH

Figure 1 shows the proposed programming model and execution architecture. Note the strong separation of concerns; offloading parallelism extraction and tuning away from the programmer to an automatic system reduces time to market of new systems and features.

### 2.1 Relaxed Sequential Programming

For many applications, there is no single required order of execution or even a single correct output. Semantically, a multitude of execution orders and outputs may be equally correct. However, a side effect of expressing the algorithm in a sequential programming model is the implicit declaration of an arbitrary single order and single output as correct. Consider the code in Figure 2. The dependency on the `seed` variable across invocations of `random` prevents a compiler from scheduling iterations of the loop in `main` to execute concurrently and without synchronization[1]. However, if the programmer marks the function as *Commutative*, indicating that invocations of `random` may happen atomically in any order, then the compiler is free to apply DOANY parallelization [13], which schedules iterations for concurrent execution and ensures consistency of `seed` state by making calls to `random` atomic. Recent work demonstrates how scalable parallelism of different types can be obtained via a generalized commutativity framework [7].

Some data structures offer sequential semantics to programs that may work correctly even with relaxed semantics. For example, in many iterative convergence algorithms, ignoring flow dependencies and allowing reads of stale values (earlier versions of a memory location) unlocks parallelism by enabling iterations of a loop to execute in parallel [12]. Respecting the dependencies improves algorithmic convergence time, but breaking the dependencies unlocks parallelism which in several cases compensates the former.

[1]Sophisticated parallelization algorithms can extract some forms of parallelism from the loop [10].

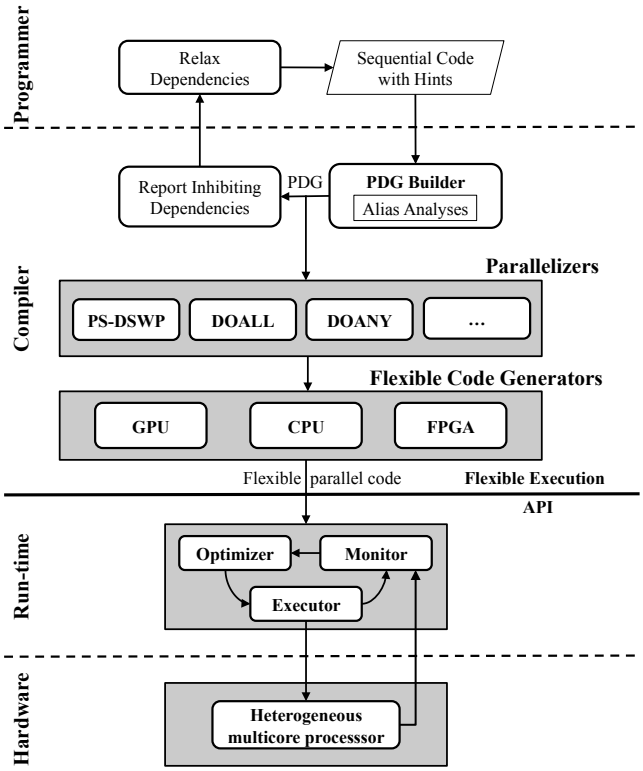


**Figure 1: Programming and Execution Architecture**

### 2.2 Parallelizing Compiler

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

Referring to Figure 1, the compiler discovers parallelism by building a Program Dependence Graph (PDG) of the hottest outermost loop nest. The compiler processes the programmer-specified order and data consistency relaxations to remove some dependencies (while inserting appropriate synchronization where necessary). The compiler then applies multiple parallelizing transforms, such as DOALL [2], DOANY [13], and PS-DSWP [10], to the PDG of a loop nest. The framework can accommodate additional, new transforms. Each transform extracts a distinct form of thread-level parallelism—data parallelism, pipeline parallelism, and task parallelism—encapsulated in code packages called *tasks*. The original, sequential version of the loop is also maintained as a task. Dynamic instances of a task may either execute sequentially or in parallel, depending on the task's dependency pattern. Note that parallel execution may involve communication or synchronization.

Flexible parallel execution entails dynamic scheduling of task instances across different types of cores, execution of parallel tasks by a dynamically varying number of cores, and pausing a set of tasks followed by resumption of a possibly different set of functionally equivalent tasks. To facili-

```
1 @Commutative
2 int random() {
3   int temp = seed / 127773L;
4   seed = 16807L * (seed - temp * 127773L)
5     - (temp * 2836L);
6   if (seed < 0)
7     seed += 2147483647L; // (2<<31)-1
8   return seed;
9 }
10 int main() {
11   for(i=0; i<N; i++) {
12     int seed = random();
13     work(seed);
14   }
15 }
```

**Figure 2: *Commutative* annotation allows multiple calls of `random` to execute out of order, unlocking loop level parallelism**

tate such execution, the compiler uses the flexible code generation algorithm [9] to generate multiple versions of code suited to the different components of the heterogeneous architecture [3], and parameterizes the code on a relatively small set of variables whose values are dynamically tuned to optimize for any given execution environment [9].

An embedded system developer knows that certain code blocks may execute efficiently on specific accelerators. The developer can share this knowledge with the compiler by tagging code blocks with affinity hints. The compiler uses the hints to suggest initial program configurations in different program phases to the run-time system.

## 2.3 Optimizing Run-time System

Workload, platform, and available resources constitute a program's execution environment. Recent work demonstrates the effectiveness of an optimizing run-time system in improving system performance by adapting program configuration to change in execution environment [9].

While deploying an embedded system, the developer specifies an optimization objective; for example, a real-time latency requirement under specified peak power constraint and average energy constraint. Given this objective, the run-time system drives the embedded program through a series of program configurations to identify the optimal configuration. A program configuration may consist of: (1) the specific set of tasks chosen to implement desired functionality; (2) the Degree of Parallelism (DoP), the varying number of cores allocated to every parallel task; and (3) the mapping of tasks to the different types of computational units available. The initial program configuration may be the one generated by the compiler using the programmer's affinity hints. The run-time system records statistics/characteristics of flexible execution and the hardware platform by using monitoring interfaces provided by the hardware. Different control and optimization techniques, both open loop and closed loop, use the gathered run-time information to converge upon the optimal program configuration [8].

Severe resource constraints in embedded systems shrink the configuration space of a flexible parallel program significantly. While this reduces the slack available to a run-time optimizer, it also reduces the size of the search space potentially leading to quicker convergence. Tight memory budget

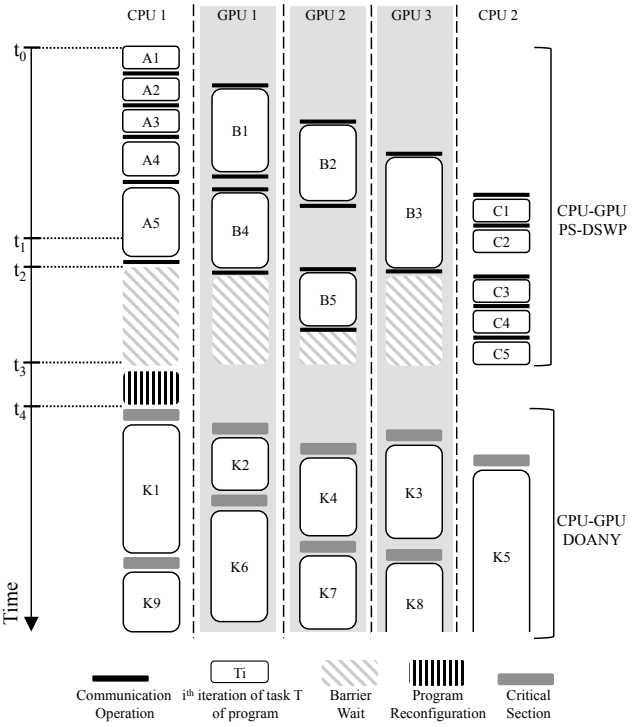


**Figure 3: Flexible Execution: Run-time system adapts program configuration including task-to-core mapping to suit execution environment**

and smaller address space may make some parallelization schemes impractical. Moreover, extreme demands for energy efficiency prevent adoption of parallelization schemes with relatively high constant overhead such as speculative parallelization. Consequently, the design of an efficient and flexible run-time optimizer for embedded systems requires additional research.

## 2.4 Flexible Parallel Execution

Figure 3 shows an example of the proposed execution model on a hypothetical machine with two CPU cores and three GPU cores. GPU core lanes are shaded grey. A parallel region consists of a set of concurrently executing tasks. (The inscription inside each box indicates the task and its dynamic instance; e.g., $B5$ represents the fifth dynamic instance of task $B$.) At time $t_0$, the program is launched with a pipeline parallel configuration (PS-DSWP [10], which splits a loop body across stages and schedules them for concurrent execution) having three stages corresponding to tasks $A$, $B$, and $C$. $A$ and $C$ are executed sequentially by the CPU cores whereas $B$ is executed in parallel by three GPU cores as determined by the run-time system. Note that a single parallel region is executed on both the CPU *and* GPU, with the appropriate code packages compiled for the CPU and GPU, respectively, being dispatched. This is in contrast to models where execution on the CPU and GPU is serialized and orchestrated statically. The run-time system measures the performance of this configuration and tries alternative configurations to avoid a local maximum. At time $t_1$, the run-time system signals the program to pause. The core receiving this signal (CPU 1) acknowledges the signal at time

$t_2$ and propagates the pause signal to the other downstream cores. At time $t_3$, the program reaches a known consistent state[2], following which the run-time system determines a new allocation of resources. At time $t_4$, the run-time system launches a data-parallel configuration (DOANY [13], which schedules loop iterations for parallel execution while synchronizing shared data accesses by means of critical sections). Task $K$ and the associated critical section implement the same functionality as tasks $A$, $B$, and $C$ combined. While the example execution shows some cores idling at a barrier prior to reconfiguration, the barrier wait can be eliminated in common reconfiguration scenarios [8].

In the flexible parallel execution model, task serialization is enforced only by the dependency structure of the algorithm and global resource hazards, and not by artificial local hazards created by the inflexibility of a programming model. For instance, in OpenMP, a parallel region and its continuation are serialized. By contrast, in flexible parallel execution, the continuation may be launched on any available computation unit, even if the unit does not match the affinity hint that may have been specified by the programmer for the continuation code block.

Finally, researchers have demonstrated that synergistic use of extensions to the sequential programming model, a parallelizing compiler, and an optimizing run-time system for flexible parallel execution, unlocks scalable parallelism and leads to efficient execution of a wide variety of programs. One study improved the response time and throughput of various benchmarks from the SPEC and PARSEC benchmark suites by 136% (geomean) over their original manual parallel implementations. The study also maximized performance under a variety of power and energy constraints. Further, in the context of a multiprogrammed system, the study improved system-wide performance while reducing energy consumption [8, 9].

# 3. CONCLUSIONS

To overcome the challenges of software development for emerging embedded system platforms, this paper describes a novel tightly integrated approach that has shown promise on mainstream computing platforms. Applications are developed in the sequential programming model (including legacy applications) and are automatically enhanced to execute flexibly on heterogeneous multicore platforms. The run-time system holistically optimizes the execution of multiple flexible parallel programs executing simultaneously.

While the effectiveness of this approach has been demonstrated on mainstream computing platforms, research is yet to be done to deploy this approach on embedded systems with their unique challenges. However, the multidimensional nature of performance objectives and constraints in the embedded domain makes the manual, static development process untenable, and makes an approach of the kind described in this paper absolutely essential.

# 4. ACKNOWLEDGEMENTS

We thank Nick Johnson for his feedback. This material is partly based on work supported by National Science Foundation Grant 1047879. Jae W. Lee was supported by the Korean IT R&D program of MKE/KEIT KI001810041244.

[2]An analysis can identify points at which the program can be efficiently paused [8].

# 5. REFERENCES

[1] The OpenMP API specification. http://www.openmp.org.

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

[3] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th ACM/IEEE Design Automation Conference (DAC)*, 2012.

[4] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2012.

[5] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 4*, April 2011.

[6] The OpenAcc API specification for accelerators. http://www.openacc-standard.org.

[7] 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.

[8] A. Raman. *A System for Flexible Parallel Execution.* PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, December 2011.

[9] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: A system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[10] 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.

[11] J. Reinders. *Intel Threading Building Blocks.* O'Reilly & Associates, Inc., 2007.

[12] A. Udupa, K. Rajan, and W. Thies. ALTER: Exploiting breakable dependences for parallelization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

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