Parkour: Parallel Speedup Estimates for Serial Programs

Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor
University of California, San Diego

Abstract

We present Parkour, a tool that creates parallel speedup estimates for unparallelized serial programs. Unlike previous approaches, it does not require any prior human analysis or modification of the program. Parkour automatically quantifies the parallelism of a given program and provides an approximate upper bound for performance, modeling fundamental parallelization constraints. For the evaluation, Parkour is applied to three benchmarks from the NAS Parallel benchmark suite running on a 32-core AMD multicore system, and three benchmarks running on the fine-grained MIT Raw processor. The results are compelling. Parkour is able to significantly improve the accuracy of parallel speedup estimates relative to the critical path analysis technique that it extends.

1 Introduction

As we seek to take advantage of the performance opportunities that multicore provides, we are also simultaneously faced with an enormous software engineering challenge. Parallelization of serial code is unpredictable for even the most expert programmers, generating great uncertainty that threatens both project feasibility and release schedules.

These challenges are especially pronounced in high-performance embedded systems that use large arrays of cores such as Tilera [6] 100-core systems, Coherent Logix's HyperX, and 512-core Nvidia Fermi chips to meet aggressive performance goals. For the purposes of planning, there are many details that we would like to know *before* we start the laborious task of parallelizing the code. Will a 64-core or 100-core Tilera chip be sufficient to attain the required speedup for my autonomous vision system? How many cores should be allocated to each component? Should a key algorithm be swapped out because it is inherently serial? What should we tell a junior engineer to shoot for in terms of speedup for a target piece of code? These are all challenging questions that could affect not only how projects are executed but also whether they are attempted at all.

```
$> make CC=parkour-cc
$> ./sha data
$> parkour --openmp
Cores     1  2  4    8    16   32   64
Speedup   1  2  3.8  3.8  3.8  3.8  3.8
(est.)
```

Figure 1: **Parkour's User Interface.** After compiling and executing the program, Parkour produces estimated upper bounds on speedups for the program.

We introduce Parkour, a parallel speedup estimation tool, to help users discover the answers to these questions. Given an unmodified, serial version of a program, and representative inputs, Parkour automatically generates approximate upper bounds on the speedup attainable for a target system. Parkour does this through a combination of dynamic and static analyses.

Figure 1 depicts the usage of Parkour in greater detail. `parkour-cc`, a drop-in replacement compiler, is used to generate an instrumented binary for the serial program. The program is then run on representative inputs, and parallelism-related instrumentation data is collected. `parkour` is then used to analyze the instrumentation data and produce speedup estimates for varying quantities of cores.

Parkour produces approximate upper bounds on parallel speedup by incorporating the parallelism-related instrumentation with machine constraints such as the number of cores, basic synchronization overhead, and constraints from parallelization systems like OpenMP. With Parkour, the user can quickly and accurately estimate the potential benefit of parallelizing a program without going through the difficult process of refactoring the code for parallelism.

Parkour's parallelism analysis is based on *critical path analysis (CPA)* [13], which dates back to the mid-80's. CPA analyzes dynamic dependencies in a program, identifying the longest dependency chain. This chain is ef-

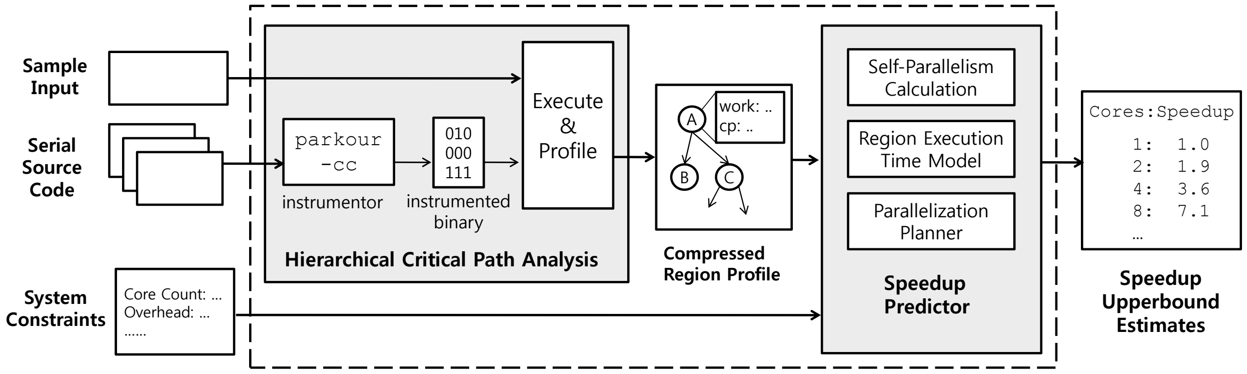

Figure 2: **Parkour System Architecture.** Starting with a program's source code, Parkour statically instruments the code to perform HCPA. Running the instrumented binary on the sample input produces a region profile that contains both the work and the critical path length for each region. Parkour's speedup predictor utilizes the profile information to provide estimated upper bounds on speedup for different core counts.

fectively the execution time on a parallel machine with infinite resources and zero communication costs. CPA suffers from two major drawbacks that hinder its utility in predicting speedup. First, its parallel execution time model is far too optimistic. This results from modeling an unrealistic machine with no execution constraints. Second, it cannot resolve parallelism within the nested region structure of a program. For instance, it cannot identify how much parallelism is attributable to the outer most loop as opposed to the innermost loop. This limitation is a result of ignoring the program's hierarchical structure.

Recent work [11, 8] showed that a number of these issues can be addressed through two key mechanisms. The first mechanism, *hierarchical critical path analysis*, or *HCPA*, measures the critical path across many nested regions. The second mechanism, the *self-parallelism* metric, provides a heuristic function that allows the parallelism levels in these regions to be effectively subtracted, providing the basis for parallelism localization. While this prior work focuses on the creation of a profiling tool, this paper shows that these techniques can be extended to estimate performance upper bounds on the parallelization of serial code, allowing software engineers to better evaluate the potential of parallelization for particular pieces of serial code.

The remainder of this paper proceeds as follow. We begin by reviewing related work in Section 2. In Section 3 we provide a high-level overview of Parkour before providing more details in Section 4. Results are presented in Section 5 before concluding in Section 6.

2 Related Work

Parallelism Profiling Critical path analysis (CPA) dates back to the Kumar's work in the late 80's [13]. This work, along with other early work [3], lays the groundwork for dynamically measuring the critical path of a program in order to determine the amount of parallelism on an unconstrained system. Other works have introduced CPA-related metrics such as *smoothability* [17] and *slack* [15] to understand how parallelism maps to constrained processors. Parkour extends CPA with hierarchical CPA and self-parallelism in order to localize parallelism to specific regions of the program rather than across the whole program. Localized parallelism allows for speedup predictions that are much more accurate.

Dependence testing is the other major approach to understanding parallelism in a program. Dependence testing is used to determine if two parts of the code have dependencies between them; if they do not then they may be executed in parallel. `pp` [14] is an early example of dependence testing, looking at the parallelism in loop nests to determine the best granularity at which to parallelize loops. More recent approaches to dependency testing include Alchemist [19] and SD3 [12]. Unlike Parkour, dependence testing approaches do not look to quantify parallelism so much as provide a binary, yes/no answer as to whether parallelism exists. Dependence testing tends to be more pessimistic than CPA, being more sensitive to program structure. As a result, it can miss nuanced forms of parallelism that may require program transformations to enable. Since Parkour is more optimistic, it can identify hidden parallelism opportunities and provide better speedup predictions, realizing that the programmer will be able to perform many of the required transformations.

Performance Prediction There have been recent efforts to predict serial performance including work by Hoste et al [10]. In theory, these predictions could be combined with Parkour's speedup predictions to predict the parallel execution time of a program

CilkView [9] and Intel Parallel Advisor [1] are recent tools with features that bear some similarity to Parkour. Like Parkour, they predict parallel performance on a target with arbitrary number of cores. Unlike Parkour, however, CilkView and Parallel Advisor rely on user annotations or code transformations to predict speedup. Parkour minimizes user effort in prediction by automatically detecting parallelism in the unmodified serial program.

Several research efforts seek to predict the scalability

```
int main() {
  foo(N);
}

void foo(int size) {
  for(i=1 to size) {
    // loop a
  }
  for(i=1 to size) {
    // loop b
  }
}
```

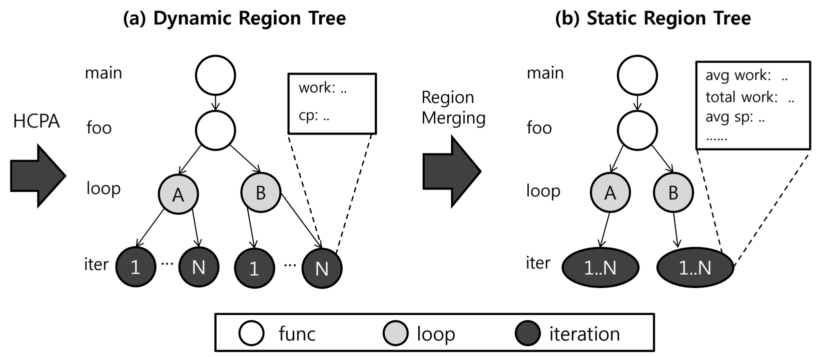

Figure 3: **Parkour's Program Representation.** (a) `parkour-cc` demarcates each function, loop, and loop iteration. Executing the instrumented binary forms a region tree consisting of dynamic regions. (b) The speedup predictor transforms the dynamic region tree into a static region tree by merging dynamic regions corresponding to the same static region, providing the program structure appropriate for speedup prediction.

of parallel programs. They start with a parallel version of the program, execute it on a small number of cores/processors, and predict how it will perform on a large number of cores/processors. Barnes et al [5] look at extrapolating performance using a number of techniques such as finding the global critical path. Zhai et al [18] rely on deterministic replay to measure the time of each part of the program on a single node, combining these single-node runs to determine the multi-node performance. These efforts are based on a pre-existing parallel implementation, unlike Parkour, which works on an unmodified serial program.

3 Parkour Overview

Parkour consists of two general phases: hierarchical critical path analysis (HCPA) and speedup prediction, which are depicted in Figure 2. In the following subsections, we will overview each of these phases in more detail.

3.1 Hierarchical Critical Path Analysis

HCPA [8] is an extension of critical path analysis that measures the parallelism in each region of the program rather than the parallelism across the whole program. HCPA has both a static component—where instrumentation is inserted into an unmodified, serial program—and a dynamic component—where the instrumented program is executed to collect parallelism statistics.

HCPA's static component utilizes a drop-in compiler replacement, `parkour-cc`, that produces an instrumented binary. The dynamic component consists of a special HCPA library—linked into the instrumented binary—that tracks dynamic data and control dependencies, while uncovering the program's structure. The HCPA runtime library uses a sophisticated shadow-memory implementation that concurrently tracks the control and data dependencies in multiple levels of the program hierarchy. HCPA tracks only true dependencies, leveraging LLVM's capabilities to avoid output dependencies and to break false-dependencies introduced through the use of induction and reduction variables. HCPA provides the critical path length and amount of work done in each nested dynamic program region.

Parkour's implementation of HCPA creates regions from every function, loop, and loop iteration; these region types were chosen because they to correspond to the unit of parallelization. Figure 3(a) shows an example of how the region structure looks during runtime. The number of dynamic regions could quickly explode when there are many nested loops. To avoid huge profile sizes, Parkour uses a dictionary-based compression technique to combine identical regions. The compression technique is highly effective, reducing the profile output size by 119,000X—to an average size of 150KB— on NPB [4].

3.2 Speedup Predictor

Parkour's speedup predictor estimates the upper bound on parallel performance. This prediction is based on both the specified system constraints and the parallelism profile data produced during HCPA.

Since programmers parallelize static regions rather than dynamic regions—e.g. using OpenMP's `parallel for` pragma to parallelize a loop—profile data should be oriented towards static regions. Parkour converts the runtime region tree into a static region tree by merging all dynamic instances of a static region into a single node as shown in Figure 3(b). Each static node will contain the average parallelism across all corresponding dynamic nodes.

Leveraging the program structure exposed during HCPA and the calculated self-parallelism, the speedup predictor employs a region execution time model to estimate the parallel execution time of each region. In addition to program structure and self-parallelism, the model incorporates other key factors that impacts parallel execution time including number of cores and parallelization overhead. We will discuss this model in more detail in the next section.

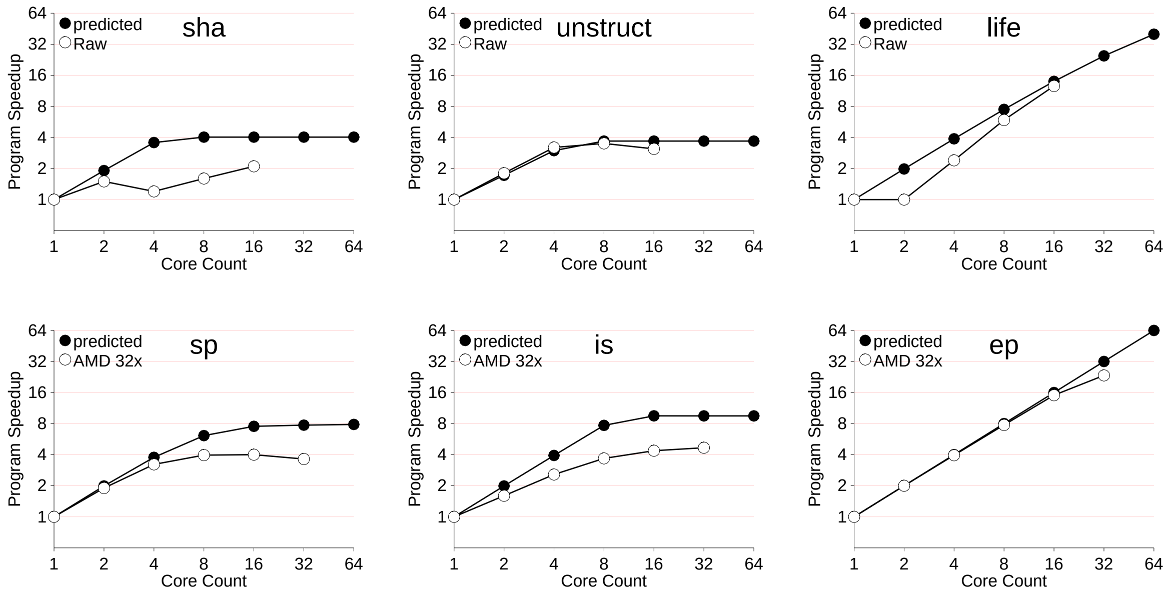

Figure 4: **Predicted and Measured Speedup.** The graphs in the upper row and in the lower row show predicted and measured speedup for Raw and Multicore platforms, respectively. In all six benchmarks on two platforms, Parkour successfully provides tight speedup upperbounds.

Given the profiled information and the execution time model, the predictor provides a speedup upper bound by finding the *parallelization plan* that minimizes the execution time. A parallelization plan abstracts how a program is parallelized by allocating available cores to each node in the static region tree. A *parallelization planner* strives to find the best parallelization plan while honoring parallelization constraints.

4 Parkour Details

In this section, we will describe in more detail how Parkour builds upon HCPA to predict parallel speedup.

4.1 Self-Parallelism Calculation

While HCPA provides the amount of parallelism in each region of a program, it does not differentiate between the parallelism that is local to that region and the parallelism that is inherited from child regions. Parkour localizes parallelism by calculating self-parallelism, a metric that estimates the maximum amount of parallel speedup obtainable from a region.

Self-parallelism subtracts the children's parallelism from region R by assuming the children have been fully parallelized. This reduces the time spent in R, thus reducing the ratio of work to critical path length. The following equation shows how the self-parallelism of R, $SP(R)$, is calculated:

$$SP(R) = \frac{\sum_{k=1}^{n} cp(child(R,k)) + SW(R)}{cp(R)} \quad (1)$$

where n is the number of children of R, $child(R,k)$ is the k^{th} child of R, and $cp()$ is the critical path length of the region. $SW(R)$ represents self-work: the total work in R less the total work in all of R's children.

4.2 Modeling Parallel Execution Time

With HCPA and self-parallelism, Parkour can begin to estimate the execution time of the program when only a subset of regions are parallelized. Parkour incorporates not only self-parallelism but also program structure, the number of cores allocated to a region, and the overhead of parallelization. Parkour calculates the execution time of region R, $ET(R)$, according to the following equation:

$$ET(R) = \frac{SW(R) + \sum_{k=1}^{n} ET(child(R,k))}{min(SP(R), A(R))} + O(R) \quad (2)$$

where $A(R)$ is the number of cores allocated to R and $O(R)$ is the target-dependent parallelization overhead.

The time model relies on the program structure in that each region's execution time is impacted not only by it's self-work but also the time spent in its children. The speedup from parallelization is limited by either the lack of self-parallelism ($SP(R)$) or the lack of allocated cores ($A(R)$), whichever is smaller.

The overhead function $O(R)$ impacts the profitable parallelization granularity; larger overhead requires a region with enough time reduction through parallel speedup to offset the overhead. For example, conventional multi-core processors have large overhead in the form of communication costs, greatly limiting the set of regions that can be profitably parallelized.

4.3 Parallelization Planner

Parkour attempts to provide an approximate upper bound on speedup by finding the allocation of parallel resources that minimizes parallel execution time. The mapping of resources to program regions is the parallelization plan. To determine the best plan, Parkour incorporates target constraints such as the maximum number of cores available, parallelizable regions, and nested parallelization. As a planner incorporates more parallelization constraints, Parkour's speedup estimation will become more accurate.

Parkour includes planners for two different platforms: "Raw" and "Multicore". The Raw platform is a tiled processor with an automatically parallelizing compiler [16], while the Multicore platform models a large x86 multicore processor system programmed with OpenMP.

The main parallelization constraint in the Raw platform is that it exploits only fine-grained parallelism that exists within unrolled basic blocks. To incorporate this constraint, the Raw planner parallelizes only unrolled versions of leaf regions in the static region tree, as they approximately represent instruction-level parallelism in basic blocks after the Raw compiler has unrolled and parallelized them.

On the other hand, the Multicore platform favors loop regions, and nested parallelization is rare due to excessive synchronization cost. Due to these restrictions on nested parallelism, either a region R or some subset of its descendant regions will be parallelized. Parkour uses its parallel execution time model to determine whether it is more profitable to parallelize R or its descendants. This decision requires a bottom-up approach, starting with the childless leaves and moving up the tree. The decision whether to parallelize a region is based on whether the execution time is lower when that region or when its descendants are parallelized.

5 Experimental Results

Methodology To see the effectiveness of Parkour, we compared Parkour's estimated speedup against actual speedup on Raw and multicore platforms. For the Raw platform, we selected three benchmarks from [16] and compared the estimated speedup with numbers reported in the paper. For multicore platform, we used NAS Parallel Bench [4] with 'W' input to get speedup estimation and measured the speedup with third-party parallelized version [2] on actual 32-core (8 X Quad-core AMD 8380) system. Parallelization overhead for Raw is specified as $3+2*logN$ cycles, modeling Raw's end-to-end barrier cost; the overhead for multicore is measured on the target 32-core system with microbenchmarks [7], which ranges from 2k cycles for two cores to 30k cycles for 32 cores.

Platform	Bench	CPA	Parkour	Ratio
Raw	sha	4.8	4.0	1.2
	unstruct	3447	3.7	934
	life	116278	40.0	2906
Multicore	sp	189928	7.9	24195
	is	1300216	9.5	136865
	ep	9722	63.9	152
Geomean		12905	11.9	1085

Table 1: **Speedup Prediction Comparison Between CPA and Parkour (64 core).**
CPA provides unrealistic speedup estimates as it cannot incorporate parallelization constraints. As HCPA's region profiling enables the incorporation of those constratins, Parkour reduces the speedup upperbound by 1085X on average.

Discussion We show predicted and measured speedups on Raw and Multicore in Figure 4. The upper row shows results on Raw, while the lower row shows results on the multicore. Parkour performs well across the spectrum—from embarassingly parallel ep running on multicore to very low parallelism sha and unstruct running on Raw. These results strongly suggest that Parkour's speedup prediction is effective.

Table 1 compares the predicted speedup from CPA and Parkour (64 core). As expected, CPA prediction results are far from practical use, except for a very simple kernel, sha. Parkour's prediction provides much tighter speedup upperbounds, by 1085X on average.

6 Conclusion

In this paper we have presented Parkour, a tool for estimating the parallel speedup of serial programs. Parkour automatically identifies the parallelism existing in a program and combines this with parallelization constraints to provide an upper bound for the parallel speedup on a specified system. Our preliminary results on six benchmarks and two classes of machines (AMD Opteron and a tiled processor) demonstrate Parkour's effectiveness at providing accurate upper bounds across diverse programs and machine architectures.

Acknowledgement

This research was funded by the US National Science Foundation under CAREER Award 0846152, and under Awards 0725357 and 1018850.

References

[1] "Intel Parallel Advisor 2011." `http://software.intel.com/en-us/articles/intel-parallel-advisor`.

[2] "NAS Parallel Benchmarks 2.3; OpenMP C." `www.hpcc.jp/Omni/`.

[3] T. Austin, and G. S. Sohi. "Dynamic dependency analysis of ordinary programs." In *ISCA '92: Proceedings of the International Symposium on Computer Architecture*, 1992.

[4] Bailey et al. "The NAS parallel benchmarks." In *SC '91: Proceedings of the 1991 conference on supercomputing*, 1991.

[5] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. "A regression-based approach to scalability prediction." In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, 2008.

[6] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect." In *IEEE Solid-State Circuits Conference*, Feb. 2008.

[7] J. M. Bull, and D. O'Neill. "A microbenchmark suite for openmp 2.0." *SIGARCH Comput. Archit. News*, December 2001.

[8] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor. "Kremlin: Rethinking and rebooting gprof for the multicore age." In *PLDI*, 2011.

[9] Y. He, C. Leiserson, and W. Leiserson. "The Cilkview Scalability Analyzer." In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, 2010.

[10] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. "Performance prediction based on inherent program similarity." In *PACT '06: Parallel architectures and compilation techniques*, 2006.

[11] D. Jeon, S. Garcia, C. Louie, S. Kota Venkata, and M. Taylor. "Kremlin: Like gprof, but for Parallelization." In *Principles and Practice of Parallel Programming*, 2011.

[12] M. Kim, H. Kim, and C.-K. Luk. "SD3: A scalable approach to dynamic data-dependence profiling." *Microarchitecture, IEEE/ACM International Symposium on*, 2010.

[13] M. Kumar. "Measuring parallelism in computation-intensive scientific/engineering applications." *IEEE TOC*, Sep 1988.

[14] J. R. Larus. "Loop-level parallelism in numeric and symbolic programs." *IEEE Trans. Parallel Distrib. Syst.*, 1993.

[15] L. Rauchwerger, P. K. Dubey, and R. Nair. "Measuring limits of parallelism and characterizing its vulnerability to resource constraints." In *Proceedings of the 26th annual international symposium on Microarchitecture*, MICRO 26, 1993.

[16] Taylor et al. "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams." In *ISCA '04: International Symposium on Computer Architecture*, Jun 2004.

[17] K. B. Theobald, G. R. Gao, and L. J. Hendren. "On the limits of program parallelism and its smoothability." In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, 1992.

[18] J. Zhai, W. Chen, and W. Zheng. "Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node." In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, 2010.

[19] X. Zhang, A. Navabi, and S. Jagannathan. "Alchemist: A transparent dependence distance profiling infrastructure." In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.