

# A Platform for Competitive Execution

Oliver Trachsel, Christian Fischlin, and Thomas R. Gross

Department of Computer Science

ETH Zurich

Zurich, Switzerland

{oliver.trachsel,thomas.gross}@inf.ethz.ch, cfischli@student.ethz.ch

**Abstract**—We propose competitive execution as an approach to leverage multi-core systems for sequential programs. Different variants of a program are executed competitively in parallel on multiple processor cores. Variants are generated by selecting different optimization strategies during compilation. The execution of a program is split into phases, and the variants compete in each phase. The fastest variant determines the execution time of the phase, thereby reducing the overall execution time of the program.

This paper presents a software platform that enables competitive execution under Linux. The software system orchestrates the execution of the variants and performs state synchronization between variants. The usage of the system requires minimal modifications to existing programs, in the form of a set of system calls to define execution phases. We also propose a language-integrated extension of the competitive execution paradigm to define variant-selection based on metrics other than performance.

## I. INTRODUCTION

In contrast to the increasing trend towards multi-core and many-core systems, many programs to be executed on these platforms offer only a limited degree of concurrency or are even inherently sequential. They therefore cannot directly leverage multiple processor cores.

We investigate competitive execution, an approach to let such sequential programs benefit from multi-core computer systems. The idea of competitive execution is to execute multiple *variants* of the same program (or of parts of the program) in parallel on different cores and to ensure that the program progresses at the rate of the fastest variant. The execution of a program is split into phases, and different variants compete in each phase. The fastest variant determines the execution time of a phase. We generate variants

of a sequential program by selecting different optimization strategies during compilation. Other approaches to generate different variants may be used, such as providing different implementations for specific program phases, e.g., based on different algorithms or heuristics.

In this paper, we present a software platform that offers support for competitive execution (or *CPE*, for *competitive parallel execution*). We describe the general architecture of the prototype implementation and the mechanism to enable competitive execution for existing programs. The software platform is integrated into the Linux kernel. Existing programs require minimal modifications to use the platform, in the form of a set of system calls to define execution phases.

Furthermore, we propose a programming language extension based on the competitive execution concept that enables programmer-controlled competitive execution and allows for variant-selection based on metrics other than execution performance. E.g., it is possible to compare the solution quality of multiple algorithms for a specific problem and to pick the variant that provides (within some time quantum) the best result. The language extension and associated execution semantics can be used to easily add partial parallelism to sequential programs, without the need to guarantee thread-safety.

This paper is a first presentation of how to use the idea of competitive execution in the context of multi-core machines, which provide (or will provide) ample computation resources. Our ongoing research focuses on a detailed evaluation of the software platform. There are a number of issues that we attempt to address: We want to study the inherent overhead imposed by the approach and consequently identify the optimal (or a reasonable) granularity for competitive stages.

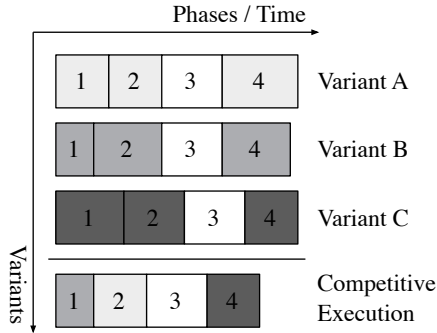


Fig. 1. Competitive execution overview.

Stage granularity is analyzed with respect to the overhead induced by the platform, by program properties such as the memory load, and by the performance differences between program variants. Many additional issues should be investigated. E.g., further study is also needed on how competitive execution impacts the usage of shared resources such as memory bandwidth or shared caches. And of course there is the issue of dealing with energy consumption: if the option is running multiple variants vs. running a single variant and turning off the extra cores, then there is a trade-off between the additional power consumption (needed to execute multiple variants instead of a single default variant) and the reduced execution time.

The remainder of this paper is organized as follows. Section II explains the ideas and main concepts behind competitive execution. Section III describes the architecture of our CPE software platform integrated in the Linux kernel and Section IV presents some preliminary performance results. Section V proposes a programming language extension based on the idea of competitive execution, Section VI describes some experiences and lessons learned during implementation, Section VII gives an overview of related work, and Section VIII concludes the paper.

## II. COMPETITIVE EXECUTION

This section describes the general concept behind competitive execution. The starting point is a sequential program consisting of one or more execution phases. Multiple candidates (called *variants*) are generated for a set of these phases (i.e., variants perform the tasks of the phases), each of which has the potential to perform best under a given execution setting. As an example,

the performance of a variant may depend on actual input data used at run-time or on the characteristics of the platform the program is executed on. A competitive execution approach allows the different variants to compete in each phase. Execution proceeds to the next phase as soon as one variant completes a phase. The goal of the approach is that the complete execution time of the program approximates the sum of the execution times of the fastest variants for each phase. Fig. 1 illustrates the idea based on a program with four execution phases and using three variants. Variants exist for all but the third phase.

We apply this execution approach in the context of multi-core systems. Variants are competitively executed on different processor cores in a controlled manner. The effects of the execution of a variant-based program is not distinguishable from the execution of the corresponding stand-alone program without variants.

Fig. 2 illustrates the control flow of a competitive execution. The execution alternates between *sequential phases*, where only a single variant is running, and *competitive phases*, where multiple variants are running in parallel. The competitive phases can in turn be divided into *stages*. Variants compete against each other in every stage, after which the platform synchronizes the program state of the winning variant with all its peers and initiates the succeeding stage.

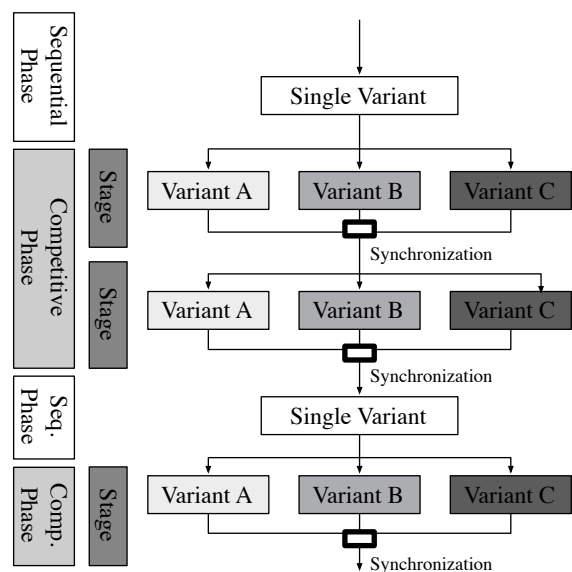


Fig. 2. Competitive execution control flow.

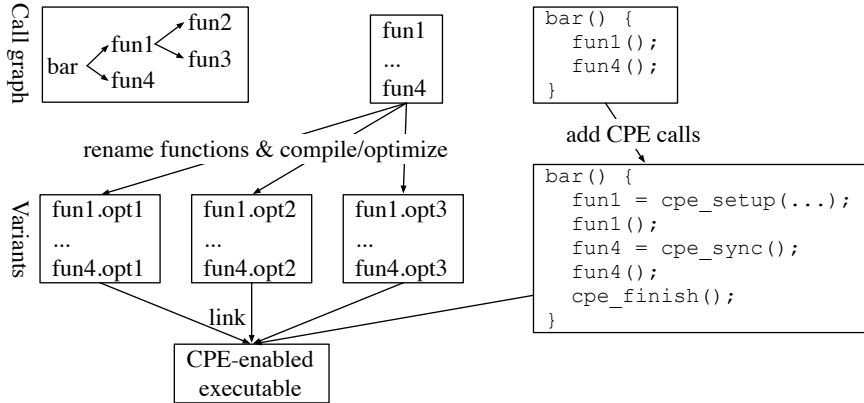


Fig. 3. Generation of a CPE-enabled executable.

### III. PLATFORM ARCHITECTURE

This section describes the architecture of the CPE (*competitive parallel execution*) software platform and how it is integrated into the Linux kernel. The current implementation is based on Linux version 2.6.22. The OS-integrated CPE platform provides a mostly autonomous way to apply the competitive execution concept to existing sequential programs.

We chose to integrate our CPE platform into the operating system for two main reasons: 1) control of process scheduling can be leveraged to coordinate the execution of variants, and 2) access to the the virtual memory subsystem can be leveraged to synchronize state between variants.

#### A. OS interface

The interface to the platform is realized using three system calls, which are used to augment sequential programs to enable competitive execution:

- `cpe_setup` configures and initiates a competitive execution phase. Pointers to variant functions for the different stages are provided as an argument. The effect of this system call is similar to `fork` in that it results in multiple variant processes being spawned. It returns a variant-specific function pointer representing the first competitive stage.
- `cpe_sync` defines the boundary between two stages. The first variant invoking this system call after a stage is declared stage winner. Its peer variants are stopped and its state is synchronized with all variants. Execution then proceeds with the next competi-

tive stage. The system call returns a variant-specific function pointer, representing the succeeding stage.

- `cpe_finish` defines the end of a competitive execution phase. Execution proceeds sequentially in the context of the first caller. All other variants are stopped and removed from the system.

#### B. Variant generation

Fig. 3 illustrates the process of generating a CPE enabled version of a sequential program, including variant generation and combination. Variants are provided in the form of functions and are differentiated using function renaming. A set of functions defining one variant is generated using a specific compiler optimization strategy. The whole program including all sequential parts and all variants is combined into a single executable. This ensures that the memory layout for global data is the same for all variants and thereby simplifies and accelerates program state synchronization.

Variant generation and function renaming are currently performed manually, using a small set of optimization flags. We plan to combine an offline approach to statically select a set of good candidate optimization strategies and let these candidates compete at run-time using our platform. Also, the whole process of generating variants and creating the executable can be automatized.

#### C. Example

Listing 1 shows a simplified program snippet demonstrating the usage of the platform interface. Lines 2–6 configure three stages with

two variants each, by setting the pointers of the functions representing the stage/variant tuples. Execution is sequential until line 10 where the `cpe_setup` system call is used to initiate competitive execution. Lines 15 and 17 initiate a state synchronization between the variants. The calls to `cpe_setup` and `cpe_sync` return variant-specific functions, representing the succeeding stage; all other code between the call to `cpe_setup` and `cpe_finish` (which completes competitive execution) is common to all variants.

In its current state, the platform requires that programs are manually annotated using these system calls. This process can be automatized and integrated into the compilation process of a program.

#### D. Process coordination

The winning variant of an execution stage is the one which performs the first call to `cpe_sync` or `cpe_finish`. It sends an interrupt to all other variants to inform them that the stage has been completed. In addition, a CPE-process checks the state of the other variants at three points of control: 1) when entering the kernel upon invoking a system call; 2) upon completion of a system call, just before returning to user-space code; and 3) when entering the process scheduler.

---

```

1 cpe_example() {
2   cpe_data call_pointers = malloc(...);
3   call_ptrs.var[0].stage[0] = fun1_opt1;
4   call_ptrs.var[1].stage[0] = fun1_opt2;
5   (...)
6   call_ptrs.var[1].stage[2] = fun3_opt2;
7
8   // Init competitive execution (2 variants,
9   // 3 stages) and start fun1 variants.
10  fun1 = cpe_setup(2, 3, &call_pointers);
11  fun1();
12
13  // Sync. program state upon completion of
14  // 1st variant and start fun2/3 variants.
15  fun2 = cpe_sync();
16  fun2();
17  fun3 = cpe_sync();
18  fun3();
19
20  // Complete competitive execution and
21  // proceed sequentially.
22  cpe_finish();
23 }

```

---

Listing 1. CPE platform interface usage example

Upon detecting that a variant has completed the current stage, the other variants stop and mark themselves as being ready for synchronization. The winning variant then performs state synchronization of CPU registers and memory contents. State synchronization is described in more depth in the following section.

#### E. Process synchronization

After each execution stage, the program state of all variants is synchronized with the one of the stage winner. Memory synchronization is performed at the level of virtual memory pages. The winner variant is the source of the synchronization and the runner-up variants are the synchronization targets. For each source/target pair, the synchronization mechanism determines the set of pages that have been modified in either the source or target variant, using the hardware-provided dirty flag associated with each memory page.

The target's page table is then modified to reflect the one of the source, and pages are copied using the copy-on-write mechanism. The actual memory-to-memory transfer to duplicate a page is performed only if either variant actually modifies the shared page during the subsequent program execution.

The memory layout of global data is identical among all variants, because they are linked to a single executable with a single shared address space. This means, e.g., that global variables are semantically shared among variants and lie at specific addresses, even though at run-time each variant operates on separate copies. This shared address space facilitates memory synchronization because no transformation code is needed to update the program state between variants.

Additionally, no transformation between stack frames is needed, because a stage is always completed inside a function that is identical for all variants, after returning from a variant-specific function. Such stack frame transformation would be necessary if stage boundaries could lie inside variant-specific functions, because their stack layout may differ.

In the current implementation, state synchronization between all variants is performed by the stage winner, leading to a *push-based* synchronization strategy. We are extending the scheme

with an alternative *pull-based* approach and intend to compare the two. In the pull-based approach the runner-up variants are responsible for synchronizing their state from the stage winner.

#### F. System call filtering

The execution of a program under the CPE platform must not be distinguishable from an execution of the original sequential program. As a consequence, I/O operations must only be executed a single time, even in the presence of multiple variants.

We use a filtering mechanism to prevent variants from performing an I/O-related system call multiple times: the first variant invoking a system call is declared to be the stage winner and is allowed to execute this and all succeeding system calls up to the following synchronization point. All other variants are stopped upon performing the same system call to avoid interference. All variants are re-synchronized and restarted at the following synchronization point. This mechanism guarantees that every system call is invoked only once.

Another approach to guarantee singular execution of system calls would be to maintain a cache of system call return values. Upon the first invocation, the system call is executed normally and its return value is cached. Upon subsequent calls, the cached value is directly returned. This approach would avoid the penalty of pausing all but one variant and execution could remain competitive even in the presence of system calls. We plan to further study the feasibility (which we expect to be dependent on the system call type) and practicability of this approach.

### IV. EVALUATION

This section presents some preliminary evaluation results obtained using a synthetic benchmark program. The goal of the benchmark is to allow a first look at the overhead of our competitive execution platform and to determine reasonable stage granularities. The benchmark allows us to study the performance of the platform depending on different program properties, including the memory load and the performance differences between program variants.

Table I shows the execution times for a set of benchmark configurations. All measurements were taken on an 8-core system with two Intel

Xeon Quad Core processors running at 3 GHz. The execution times are averaged over three program runs, and the corresponding standard deviation is indicated by a  $\pm$  sign. The benchmark runs either one, two, or four almost identical variants in two competitive stages. Each stage performs the same CPU-bound computation in a loop and writes to a number of memory pages.

The execution time of the loop and the number of modified memory pages are parametrized (columns *Iterations* and *Pages Modified*, respectively). Ten million loop iterations correspond to an execution time of approximately 100 milliseconds. The execution time of the first variant is additionally controlled by a scaling factor (*Scale VI*). A scaling factor of 0.9 means that the first program variant executes only 90% of the defined loop iterations, making it approximately 10% faster than the other variants. All other program variants execute the full number of loop iterations.

The set of modified memory pages is identical in both execution stages. These memory pages must be synchronized with the winning variant by all other variants after the first execution stage. The table contains performance numbers for a memory load of 10, 100, and 1000 modified pages, corresponding to 40KB, 400KB, and 4000KB of data.

Competitive execution assumes that it is unknown beforehand which variant will perform best for a given execution scenario (in our synthetic benchmark the first variant always performs best, its relative performance being controlled by the scaling factor). The usage of the platform is beneficial for scenarios where the execution time using multiple variants lies between the execution time of the fastest participating variant (the first one if a scaling factor of 0.8 or 0.9 is used) and the slower variant(s).

Table I shows that competitive execution is never beneficial for 10 million loop iterations, because the execution time using multiple variants is always higher than the execution time of the slowest variant alone. E.g., for 10 modified pages, even if the fastest variant takes 80% of the other variant, the two-variant execution completes in 213 *ms*, while the slowest stand-alone variant completes in 194 *ms*.

The usage of the competitive execution platform is beneficial for all scenarios with 50 or 100 million loop iterations. E.g., for 50 million

Iterations	Pages Modified	Scale V1	1 Variant	2 Variants	4 Variants	
10,000,000	10	0.8	155.3 ms	213.4 ms	221.6 ms	
		±	0.0 ms	2.6 ms	6.0 ms	
		0.9	174.6 ms	232.0 ms	237.2 ms	
			±	0.0 ms	1.1 ms	2.1 ms
			1.0	194.1 ms	248.3 ms	261.8 ms
			±	0.0 ms	6.8 ms	5.4 ms
		100	0.8	155.4 ms	209.2 ms	219.0 ms
			±	0.0 ms	9.2 ms	0.5 ms
	0.9		174.8 ms	233.0 ms	239.2 ms	
			±	0.0 ms	1.6 ms	3.9 ms
			1.0	194.4 ms	250.8 ms	259.8 ms
			±	0.0 ms	0.5 ms	1.6 ms
	1000	0.8	157.2 ms	219.9 ms	226.5 ms	
		±	0.1 ms	2.3 ms	2.2 ms	
0.9		176.4 ms	238.9 ms	250.1 ms		
		±	0.3 ms	2.8 ms	10.3 ms	
		1.0	195.8 ms	284.5 ms	268.0 ms	
		±	0.2 ms	43.5 ms	2.7 ms	
50,000,000	10	0.8	776.1 ms	834.8 ms	936.1 ms	
		±	0.3 ms	0.5 ms	170.6 ms	
		0.9	872.8 ms	930.0 ms	936.6 ms	
			±	0.1 ms	1.2 ms	2.7 ms
			1.0	969.8 ms	1,022.5 ms	1,032.1 ms
			±	0.1 ms	5.7 ms	2.2 ms
		100	0.8	776.0 ms	834.6 ms	838.5 ms
			±	0.2 ms	1.6 ms	0.7 ms
	0.9		873.4 ms	932.4 ms	936.8 ms	
			±	0.7 ms	1.8 ms	2.4 ms
			1.0	970.2 ms	1,028.9 ms	1,033.5 ms
			±	0.7 ms	2.9 ms	2.9 ms
	1000	0.8	778.0 ms	837.9 ms	846.5 ms	
		±	0.4 ms	0.9 ms	2.8 ms	
0.9		875.2 ms	937.7 ms	943.0 ms		
		±	0.7 ms	1.6 ms	2.7 ms	
		1.0	972.1 ms	1,034.2 ms	1,042.1 ms	
		±	0.8 ms	0.4 ms	1.7 ms	
100,000,000	10	0.8	1,551.4 ms	1,606.2 ms	1,609.1 ms	
		±	0.2 ms	8.1 ms	6.2 ms	
		0.9	1,745.3 ms	1,804.0 ms	1,809.3 ms	
			±	0.2 ms	2.2 ms	2.6 ms
			1.0	1,939.6 ms	1,994.4 ms	2,000.7 ms
			±	0.1 ms	8.7 ms	2.7 ms
		100	0.8	1,551.8 ms	1,608.0 ms	1,612.1 ms
			±	0.2 ms	0.9 ms	1.8 ms
	0.9		1,745.6 ms	1,806.1 ms	1,810.5 ms	
			±	0.1 ms	1.2 ms	2.0 ms
			1.0	1,941.4 ms	1,998.8 ms	2,003.7 ms
			±	1.5 ms	2.4 ms	0.8 ms
	1000	0.8	1,553.8 ms	1,616.0 ms	1,624.5 ms	
		±	1.4 ms	1.5 ms	3.6 ms	
0.9		1,747.6 ms	2,031.2 ms	1,817.3 ms		
		±	0.2 ms	381.6 ms	0.7 ms	
		1.0	1,943.2 ms	2,005.0 ms	2,009.6 ms	
		±	1.4 ms	3.4 ms	4.3 ms	

TABLE I  
SYNTHETIC BENCHMARK EXECUTION TIMES.

iterations, 1000 modified memory pages, and an execution time of the first variant of 0.9 times the execution time of the other variants, four variants complete in 943 *ms*, while the slowest variants alone require 972 *ms* and the fastest variant requires 875 *ms*.

Table I shows that the memory load (and thereby the memory synchronization) has only minimal impact on the overhead induced by competitive execution. The execution times do increase only marginally (if at all) for an increasing number of written pages and for fixed numbers of loop iterations and variants.

It is important to note the results from Table I have not been tuned and only serve to give a first chance to assess suitable phase granularities. The preliminary performance measurements show that competitive execution starts to be beneficial with stage granularities of a few hundred milliseconds for the used synthetic benchmark. Further research is necessary to analyze if this observation holds for real-world programs as well.

## V. TOWARDS PROGRAMMER-CONTROLLED COMPETITIVE EXECUTION

Based on the competitive execution concept, we propose a programming language extension that allows for straightforward augmentation of sequential programs by alternative execution paths, without the need to ensure thread-safety. On multi-core systems, these alternative execution paths are executed in parallel based on a CPE-like platform or using a transactional memory system. This programmer-controlled competitive execution approach is intended to take the concept beyond pure performance-based competition.

The language extension provides a syntactic means for a programmer to define variants of code sections, e.g. functions. These variants perform a specific computation in different ways, e.g., based on different algorithms suited for the task or using a single algorithm with different initial parameters. Each variant works on its own view of global data, and modifications are contained within this view. The programmer also defines the criterion used to decide which variant wins the competition. An arbitrary metric can therefore be used to compare variants and select a winner, instead of just their execution speed. The effects of the selected variant are made globally visible;

the effects of the other variants are squashed. In this approach, and in contrast to performance-based competitive execution, all variants complete execution before the winning variant is determined.

Listing 2 shows an outline of a programmer-controlled CPE program. It defines two variants based on a procedure `process` by specifying two different parameters. Since the variants work on semantically distinct copies of all data, they need not provide any guarantee on the set of objects they access or modify, and they need not provide any thread-safety guarantees. The evaluation code inspects the effects of the variants and computes a quality metric within the variant-specific memory contexts (denoted by `@V1` and `@V2` in Listing 2) to decide which variant to select. The exact syntax and semantics of such inspection are open research questions.

As an example, in a compiler different graph-coloring algorithms may be used to perform register allocation, resulting in more or less spill code. Using a language extension for competitive execution, additional graph-coloring algorithms can be added to the existing compiler code in a straightforward way, without the need for the compiler writer to take measures to ensure thread-safety, even if these different algorithms access and modify shared data structures. On a multi-core system, these register allocation algorithms can be executed in parallel, and programmer-provided evaluation code can select the variant that requires the lowest number of register spills and make its modifications visible.

Programmer-controlled competitive execution can also be seen as an extension to transactional programming models (e.g., [3]). In this analogy,

---

```
language_cpe_example() {
  variants (
    V1: data.process(parameterA);
    V2: data.process(parameterB);
  ) evaluate (
    if (data.qual()@V1 > data.qual()@V2)
      select V1;
    else
      select V2;
  )
}
```

---

Listing 2. Example for programmer-controlled competitive execution.

variants correspond to transactions and execute inside their transactional memory space. Picking a winner and making its effects visible corresponds to committing the winner and aborting the other variants. But while transactional programming models (and their implementation) are often based on the assumption that conflicts between transactions are rare, our execution model is oriented towards variants that are expected to have conflicting read- and write sets and are executed concurrently on purpose. Also, transactional systems generally decide autonomously which transaction is to be committed in the case of conflicts; under programmer-controlled competitive execution, this decision is made by evaluation code provided alongside the variants. Additionally, under transactional programming models, the uncommitted transactions are usually rolled-back and re-executed, but in a competitive execution model, variants are executed only once and are either selected or not.

Our operating system-based CPE platform can be used to provide the desired semantics of such a language-based approach, which includes controlled parallel execution of variants, address space separation, and state synchronization upon winner selection. Alternatively, a transactional system could be used as a back-end. We intend to study and compare these two possibilities.

## VI. IMPLEMENTATION EXPERIENCES

During development of the CPE platform we worked mainly on a Linux installation running inside a VMware virtual machine (version 6.0.1). While this has many advantages, especially through quicker and more comfortable develop-recompile-reboot cycles required when modifying core components of the Linux kernel, we had to understand that a virtual machine is indeed different from a real one.

During development some subtle problems and bugs, mainly related to timing and process synchronization, only became apparent when switching to the real machine. Even two virtual machines by the same vendor on different host platforms revealed differences in their behavior. E.g., under a VMware virtual machine hosted on a Mac (version 1.1.1) some system call types do not seem to pass through the default kernel entry point and cannot be intercepted by our filtering mechanism. The set of system calls concerned

by this issue additionally depends on the system call interface being used. A newer, library-based interface does not work for any system calls, and an older interface (where the compiler inserts system calls directly in the generated code) works for at least a set of system calls. In contrast, the same mechanism works for all encountered system calls under the VM hosted on Linux.

The lesson we learned is not to trust the virtual machine and regularly switch to a real machine for testing.

## VII. RELATED WORK

Cho [6] exploits idle workstations in a network by competitively executing distributed applications as background processes. Our approach is conceptually similar to this one but targets a single multi-core system rather than a set of networked machines.

Diniz and Rinard [7] proposed *dynamic feedback*, a technique where different code variants are selected dynamically by alternating between sampling and production phases during execution. Competitive execution has similar goals as dynamic feedback, but does not require separate sampling phases and selects the best suited variants directly during productive program execution.

The selection of an optimal optimization strategy at compile-time has been studied extensively for static and dynamic compilation settings [1], [2], [4], [5], [13], [14]. Offline selection of good candidate optimization strategies can be used to generate variants of a sequential program to be executed using our CPE platform.

We study competitive execution as an additional potential approach to obtain parallelism for programs that do not exhibit explicit data- or task parallelism. Competitive execution may be used in combination with other approaches, such as automatic parallelization [10] or thread-level speculation [8], [9], [11], [12].

## VIII. CONCLUDING REMARKS

We have presented the concept of competitive execution and a platform that integrates the concept into the Linux operating system. The paper also discussed a programmer-controlled approach to competitive execution and proposed initial ideas of a programming language extension to offer such an approach.

The current implementation exists as a prototype and we are currently evaluating and extending the CPE software platform. We hope that competitive execution will provide a valuable approach to leverage multi-core systems for sequential programs.

#### ACKNOWLEDGMENTS

We thank Susanne Cech Previtali and the anonymous reviewers for providing helpful feedback on this paper.

#### REFERENCES

- [1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. *CGO '06*, pages 295–305.
- [2] G. Bashkansky and Y. Yaari. Black box approach for selecting optimization options using budget-limited genetic algorithms. *SMART '07*.
- [3] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. *PLDI '06*, pages 1–13.
- [4] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. *CGO '07*, pages 185–197.
- [5] J. Cavazos and M. O'Boyle. Method-specific dynamic compilation using logistic regression. *OOPSLA '06*, pages 229 – 240.
- [6] S. H. Cho. Competitive execution: a method to exploit idle workstations. In *Proc. Intl. Conf. Parallel and Distributed Systems*, pages 382–391, 1997.
- [7] P. Diniz and M. Rinard. Dynamic feedback: an effective technique for adaptive computing. *PLDI '97*, pages 71–84.
- [8] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *ASPLOS '98*, pages 58–69.
- [9] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *PLDI '07*, pages 211–222.
- [10] G. Lee, C. P. Kruskal, and D. J. Kuck. An empirical study of automatic restructuring of nonnumerical programs for parallel processors. *IEEE Trans. Comput.*, 34(10):927–933, 1985.
- [11] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. *PPoPP '06*, pages 158–167.
- [12] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *ISCA '00*, pages 1–12.
- [13] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. *CGO '03*, pages 204–215.
- [14] H. Wu, E. Park, M. Kaplarevic, Y. Zhang, X. Li, and G. Gao. Dynamic optimization option search in GCC. *GCC Developers' Summit*, pages 165–174, 2007.