Variant-based Competitive Parallel Execution of Sequential Programs

Oliver Trachsel Thomas R. Gross

Laboratory for Software Technology
Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Competitive parallel execution (*CPE*) is a simple yet attractive technique to improve the performance of sequential programs on multi-core and multi-processor systems. A sequential program is transformed into a CPE-enabled program by introducing multiple *variants* for parts of the program. The performance of different variants depends on runtime conditions, such as program input or the execution platform, and the execution time of a CPE-enabled program is the sum of the shortest variants.

Variants compete at run-time under the control of a CPE-aware run-time system. The run-time system ensures that the behavior and outcome of a CPE-enabled program is not distinguishable from the one of its original sequential counterpart. We present and evaluate a run-time system that is implemented as a user-space library and that closely interacts with the operating system.

The paper discusses two strategies for the generation of variants and investigates the applicability of CPE for two usage scenarios: i) computation-driven CPE: a simple and straightforward parallelization of heuristic algorithms, and ii) compiler-driven CPE: generation of CPE-enabled programs as part of the compilation process using different optimization strategies. Using a state-of-the-art SAT solver as an illustrative example, we report for compiler-based CPE speedups of 4–6% for many data sets, with a maximum slow-down of 2%. Computation-driven CPE provides super-linear speedups for 5 out of 31 data sets (with a maximum speedup of 7.4) and at most a slow-down of 1% for two data sets.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms

Design, Experimentation, Performance

CF'10, May 17–19, 2010, Bertinoro, Italy.

1. INTRODUCTION

Many programs that are executed on modern multi-core and multi-processor systems are sequential, and thus cannot exploit the parallel features of the hardware they are running on. This paper presents competitive parallel execution (*CPE*), an approach to leverage multi-processor and multi-core systems for the execution of sequential programs. CPE is a technique for modifying and executing existing sequential applications to increase their performance on parallel systems.

The central idea of CPE is to facilitate the introduction of multiple *variants* for parts of a program, where different variants are suited for different run-time conditions. Dynamic factors such as input data or the characteristics of the platform the program is executed on determine which variant executes the fastest. Instead of trying to determine a single “best” strategy offline before program execution, a program contains different variants that compete against each other at run-time. The purpose of creating variants is to make the program adaptive to variable run-time conditions. A CPE-aware run-time system is responsible for orchestrating the execution of the variant-augmented program on the available processor cores. The goal is thereby that the program progresses at the rate of the fastest variant for each execution phase.

In this paper we present two strategies for the application of competitive parallel execution, *computation-driven CPE* and *compiler-driven CPE*. With *computation-driven CPE*, the variants to be executed are present in the program. Computation-driven CPE can, e.g., easily be employed to perform a simple and straightforward parallelization of heuristics-based search algorithms. As a representative example we create a CPE-enabled version from an existing state-of-the art SAT solver by modifying only a few lines of code of the original sequential implementation.

Compiler-driven CPE exploits the fact that many optimizing compilers are unable to identify the best optimization settings for many programs. Compiler-driven CPE therefore employs the compiler to generate variants for frequently executed parts of the program by applying different promising optimization strategies upon compilation.

We present a simple API to enable competitive parallel execution that supports both computation-driven CPE for a programmer or compiler-driven CPE for current compilation systems. The paper also presents the architecture and implementation of a CPE-aware run-time system that leverages the UNIX process model. The run-time system is realized as a user-space library that is closely tied to the operating system.

The CPE concept is evaluated using a modern SAT Solver for both, a computation-driven approach and a compiler-driven approach. The evaluation shows that a simple program modification to enable competitive execution provides super-linear speedups up to over 7x for some data input sets for a 2-way parallel execution of the program. A compiler-driven approach to CPE leads to a performance improvement in the order of 4–6% for many of the considered benchmarks.

The main contributions of this paper are the following:

- presentation of competitive parallel execution (CPE) as a technique to leverage parallel systems for the execution of sequential programs;
- description of a simple and straightforward API to enable competitive parallel execution for existing sequential applications;
- proof-of-concept that providing CPE semantics is feasible using a pure software-based approach, with reasonable complexity;
- evaluation of competitive parallel execution using a modern SAT solver.

The remainder of the paper is structured as follows: Section 2 discusses the CPE model in more depth. Section 3 discusses the architecture of our run-time system and its integration with the operating system. Section 4 presents the evaluation. Section 5 presents related work and Section 6 concludes the paper.

2. VARIANT-BASED COMPETITIVE PARALLEL EXECUTION

Competitive parallel execution (*CPE*) is a model to adapt and execute existing sequential programs to increase their performance on multi-processor and multi-core systems. The fundamental idea of CPE is to include variants of one or multiple regions of a sequential program and to let these variants compete at program execution time.

Variants of program regions may come in many forms, e.g., as variations in algorithm heuristics, by using different starting conditions, by employing different algorithms to solve a given problem, or by selecting different optimization strategies at compile time. The motivation behind introducing variants is to enable automatic adaptiveness of the sequential program to different run-time conditions. For each execution phase of the program, for which different variants exist, one of these variants will perform best. Dynamic factors such as input data or the characteristics of the execution platform determine the best (i.e., fastest) variant.

Figure 1 illustrates the general execution model of a CPE-enabled program with an example. The execution alternates between *sequential phases*, where only a single variant is running, and *competitive phases*, where multiple variants are running in parallel. The example program in Figure 1 executes two sequential and three competitive phases. Variants compete against each other in every competitive phase. At the conclusion of a competitive phase the program state of the winning variant is synchronized with all its peers. The execution then proceeds to the succeeding competitive or sequential phase.

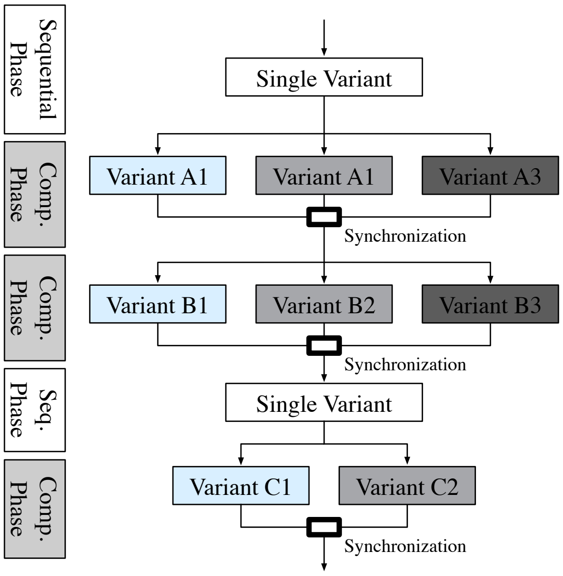

Figure 1: Example execution control flow of a CPE-enabled program with two sequential and three competitive phases. Two or three variants compete in each competitive phase. A competitive phase ends upon completion of a variant, and the program state is synchronized to the state of this winner.

This execution model is similar to a standard fork-join model, with the important distinction that upon the first arrival of a single variant at a synchronization point the other variants are stopped and synchronized with the winner, rather than awaiting the completion of all competing variants.

The behavior and semantics of a CPE-enabled program must not be distinguishable from a sequential execution, in which only a single variant runs in each phase. To this end, a CPE-aware run-time system orchestrates, monitors, and controls the execution of the program and its associated variants. The run-time system must provide two isolation properties to guarantee the semantical equivalence with the original sequential program:

1. The effects of a variant must be contained with respect to competing variants. A change in program state of one variant must thus not be observable by other variants.
2. The set of I/O operations performed by the CPE-enabled program and the order in which they are performed must not have any side-effects that differ from a sequential execution of the program.

These two isolation properties provided by the CPE model enable a simple and straightforward transformation of an existing sequential program into a CPE-enabled program. In contrast to hand-parallelizing, e.g., by introducing multi-threading, enabling competitiveness does not require any reasoning about data sharing, data races, mutual exclusion, deadlocks, or other aspects that make parallel programming intrinsically hard. The semantic model guarantees that a competitive parallel execution of a program is correct if any sequential execution (with one single variant executed for each competitive phase) corresponds to a desirable program behavior.

A central aspect of CPE is the generation of variants that have the potential of resulting in a performance gain when being executed under the CPE model. In this paper, we consider two different approaches to transform an existing sequential program into a CPE-enabled program:

1. *computation-driven competitiveness* – where variants are specified as part of the program. These variants correspond, e.g., to different implementations for specific program phases, based on different algorithms or heuristics;

2. *compiler-driven competitiveness* – where variants of parts of a program are generated by selecting different optimization strategies during compilation.

The following two sections describe these two approaches in more detail.

2.1 Computation-driven CPE

In the computation-driven approach to CPE, variants are specified by augmenting the original sequential program. Annotations, or extra code, specify the program parts where competitive execution should take place and characterize how the variants for these competitive parts diverge from each other. CPE thereby provides a simple means to enable the concurrent exploration of different possible execution paths for parts of a program.

As a consequence of the isolation properties provided by the CPE model, the process of enhancing a sequential program is a straightforward process, without jeopardizing correctness. First, the process does not require any reasoning about data sharing, data races, dead-locks, and other difficulties intrinsic to concurrent programming, because variants are guaranteed to execute in isolation with respect to each other. Second, the process does not require detailed knowledge of the inner-workings of the original program with all its data structures and algorithms or even used program libraries, because of the same isolation property and because the CPE model guarantees that any potential I/O operations are contained by the CPE-aware run-time system if they might have side-effects that would deviate from a sequential execution. Due to these properties the CPE-enabled program and its variant-specific code can be treated as a sequential program without the need of worrying about concurrency issues.

Depending on what mechanisms are used in an actual implementation of the CPE model, unforeseen I/O operations during competitive execution phases may actually nullify a potential performance benefit from using CPE by reverting to sequential execution. But the execution model guarantees that correctness is not compromised in favor of potential performance gains.

Example applications for computation-driven CPE include are heuristic algorithms, which we also consider in the evaluation of the approach. For these kind of problems, different heuristics tend to work best depending on the input data set the algorithm is executed on. A CPE-enabled program that simultaneously explores the search space with different promising heuristics—and that is thus well adapted to data sets with different characteristics—can be created with minor modifications to the original program. All that is needed is defining the heuristics parameter each variant should use, as well the points in the program where competitive execution takes place.

2.2 Compiler-driven CPE

Modern compilers offer a high number of optimization flags. E.g., GCC version 4.3.3 has over 140 distinct program options that control optimization. Although compilers generally provide default optimization strategies that deliver reasonable results on average, the optimal set of optimizations depends not only on the program being compiled, but also on the actual input data used. Such a dependence of the effectiveness of different program optimizations on input data characteristics is observed even for small benchmark kernels representing common embedded computing algorithms [10].

The compiler-driven approach to CPE builds on this observation. In this approach, variants are generated as part of the compilation process, by applying different, potentially well-suited optimization strategies when compiling the program or parts of the program. The original program is slightly adapted to enable competitive parallel execution of these variants for specific parts of the program. The concrete steps involved in creating a CPE-enabled program from an existing sequential program are the following:

1. Determine a set of potentially good optimization strategies for either the whole program or for frequently executed parts. The selection of good optimization strategies is in itself a widely studied subject and is not part of our research. Many approaches are discussed in the literature, e.g., [2, 3, 11, 16, 21, 24]. These approaches determine good optimization settings for a specific program with a specific data input set and running on a specific platform. CPE enables the automatic selection of the best among multiple such optimization settings depending on the actual program input and execution phase.

2. Collect profile information of the original program for different input data sets to determine those parts of the program that contribute most to its execution time. Introducing competitiveness for these program parts potentially leads to the highest performance gain. If the "hot" parts of a program cannot be identified based on the available data input sets, then standard heuristics may identify parts of the program as candidates for the generation of variants.

3. Perform a source-to-source transformation of the program with two modifications: i) create variants as clones of the program parts (e.g., represented by a set of functions) for which multiple variants are to be generated, and ii) specify the locations where competitive phases begin and end, along with the variants for each competitive phase.

4. Compile the program by applying the optimization strategies determined in step 1 to the variants and link all parts together to create the CPE-enabled program.

2.3 Nested competitiveness

The CPE model naturally supports nested competitiveness and is not restricted to flat competitive phases. Figure 2 illustrates nested competitiveness with an example.

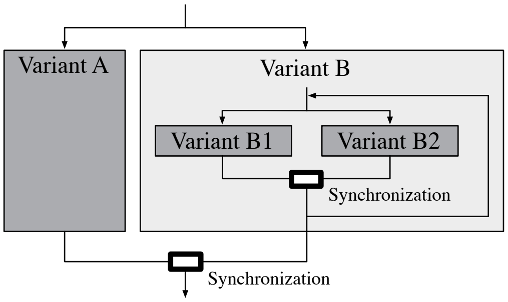

Figure 2: Example for a nested competitive execution. Variants A and B compete at the outermost nesting level. Variants B1 and B2 in turn compete in each iteration of a loop within variant B.

Variants A and B compete at the outer nesting level. Variant B in turn executes a loop where two variants $B1$ and $B2$ repeatedly compete for each loop iteration. The inner competitive execution between $B1$ and $B2$ has no effect on the execution of variant A. In turn, if A reaches the outer synchronization point before B, the latter, including potentially executing variants $B1$ and $B2$, will be interrupted, and the program execution proceeds from the state of A.

An example application of nested CPE is the introduction of more aggressive variants for parts of a given algorithm. At the outermost nesting level, the unmodified implementation, which is more conservative, competes with a modified variant, which is more aggressive but may have less predictable timing behavior. The modified variant can in turn have different variants compete for parts of the algorithm, e.g., for loop iterations as shown in Figure 2. Such a configuration ensures that the performance characteristics of the CPE-enabled program are in the worst case close to the original program, but comparable to the characteristics of the more aggressive variants for the input sets for which their strategy is successful.

3. PLATFORM ARCHITECTURE

We have implemented a CPE software platform in the form of a Linux user-space library ([20] presents an earlier, prototype of the system, which was built as an OS-service integrated into the Linux kernel). The library leverages the UNIX process- and virtual memory model, as well as the operating system's process tracing facilities to provide the semantics of the execution model. The library is implemented in C, but is usable for any programming language that can use an API provided as a shared or static library.

The following two sections present the programming interface provided to transform a program into a CPE-enabled program, and relevant details on the architecture of the runtime library. The same programming interface can be used to implement both forms of CPE discussed in this paper, computation-driven and compiler-driven CPE.

3.1 Programming interface

The API provided to augment an existing program with CPE constructs consists of the following three functions:

```
void* cpe_start(void* v1_desc, void* v2_desc, ...);
void* cpe_sync();
void  cpe_finish();
```

These three functions define the starting points, synchronization points, and end points of competitive phases. In the execution model depicted in Figure 1, `cpe_start()` corresponds to the starting point of a competitive phase with a new CPE variant configuration; `cpe_sync()` corresponds to the synchronization point between two competitive phases with the same variant configurations; and `cpe_finish()` corresponds to the synchronization point at the end of a competitive phase, after which another competitive phase can be initiated.

The first function, `cpe_start()`, is used to initiate a competitive phase. The function takes a single value per variant as arguments. This variant-specific value may contain any arbitrary information, e.g., a pointer to a function that implements a variant-specific algorithm, a pointer to a descriptor with heuristics parameters to be used by the variant, or simply a variant identifier encoded as an integer. The behavior of the function is similar to the *fork()* function, which is used on UNIX systems to create new processes. The function returns multiple times – once in each variant – and returns the variant-specific pointer that was provided as an argument. Multiple calls to `cpe_start()` may be nested to create a nested competitive parallel execution as described in Section 2.3.

The function `cpe_sync()` defines a synchronization point between two competitive phases that use the same variant configuration. The first variant to call it during an active competitive phase is declared the winner. Its program state is used by all variants when proceeding with the next competitive phase. Again, this function returns once in each variant, and the return value is the variant-specific value that was provided as an argument to `cpe_start()`.

The third function, `cpe_finish()`, terminates a competitive phase. The calling variant is declared the winner of the competitive phase, and the other competing variants are terminated. The program proceeds sequentially from the program state of the winning variant, and another competitive phase can be immediately be initiated using `cpe_start()`.

3.2 Run-time library

The CPE-aware run-time system orchestrates the execution of variants and is responsible to provide the semantics defined by the CPE model with its two isolation guarantees described in Section 2.

Isolation between variants is achieved by representing variants as user-space processes. The library relies on the fast process creation and effective copy-on-write mechanism provided by the operating system. All variants executing the same competitive phase operate in a separate virtual address space, but the memory content of their address space is initially shared. Upon a write operation by one of the variants, a private copy of the affected virtual memory page is created upon which this variant operates from this point in time. State changes are therefore local to a variant and not observable from the outside.

To ensure that the execution of a CPE-enabled program observed from the outside corresponds to a sequential execution, variants are not allowed to perform externally visible I/O operations within the same competitive phase that would deviate from a sequential execution. To this end, a designated *master process* traces the execution of all variants using the *ptrace* facility provided by the operating system. The master process is notified whenever a variant issues a system call or receives a signal from some other process and can take appropriate measures, depending on the kind of the system call. The run-time system differentiates between three different categories of operations:

- *Unconstrained operations* are performed by system calls that do not have any effect that is directly observable from the outside. Such calls can therefore proceed without restriction. These are mostly system calls that obtain state information or that modify only process-local state. Examples are memory management functions such as *brk* and *mremap*.

- *Constrained operations* have side effects visible from the outside. To provide program semantics that correspond to a sequential execution, only a single variant of a competitive execution phase is allowed to perform such an operation and competing variants are aborted upon the first such operation; such an event leads to a premature switch from competitive to sequential execution. Examples in this category are calls that send data over a network socket or that write to some file.

- *Conditionally constrained operations* are handled either like constrained or like unconstrained operations, depending on the actual parameters to the respective system call. A memory mapping operation using the *mmap* system call, e.g., is constrained if it maps the contents of a file to some memory area, because that may lead to a later modification of the concerned file. The operation is unconstrained if the created mapping is anonymous, i.e., if the call is simply used to allocate some memory region that is not backed by a file.

3.3 Architecture summary

The CPE execution model depicted in Figure 1 is mapped to a simple programming interface that consists of three functions, `cpe_start`, `cpe_sync`, and `cpe_finish`. These functions are inserted into sequential programs to create CPE-enabled programs. This programming interface can be used to implement a computation-driven, as well as a compiler-driven approach to CPE.

The CPE-aware run-time system implements the semantic model and the isolation guarantees defined in Section 2. To provide effect isolation between variants they are represented as user-level processes with a separate virtual address space. To isolate variant isolation with respect to the outside system, the run-time system monitors and restricts the execution of system calls by variants.

4. EVALUATION

We use the SAT solver MiniSat [9] to evaluate the CPE model and run-time system. SAT is the NP-complete decision problem to determine if a boolean formula in conjunctive normal form is satisfiable or not. The task of a SAT solver is to solve the associated search problem of finding a set of variable assignments for which the boolean formula evaluates to true or else reporting that no such assignment exists.

SAT solvers have a wide applicability, e.g., in IC design, image analysis, and software engineering. They address a computational problem that it not easy to parallelize, and thus are good candidates for the CPE approach. E.g., in the SAT-Race 2008[1], the best sequential solver (MiniSat) performed better than the best parallel solver (ManySAT [13]) for 37 out of 86 SAT problems.

MiniSat is suitable for our purposes because it is a state-of-the-art SAT Solver (it was the winner of SAT Race 2006 and SAT Race 2008), is widely used in both academia and industry, and its source code is freely available. MiniSat is an object-oriented program implemented in C++.

Like many other modern SAT solvers, MiniSat is based on the DPLL [5] backtracking search algorithm and uses different search heuristics to improve performance. Computation-driven CPE is well applicable to heuristic algorithms in general. For such scenarios, CPE enables the straightforward transformation of a sequential execution with a single heuristic into a competitive execution that simultaneously applies multiple different heuristics.

All experiments are performed on 31 SAT data sets that originate from the SAT competition 2007 benchmark suite[2]. All data sets from categories *crafted* and *random/2+p* with a sequential execution time between 50s and 500s were selected. The programs are run on an Intel Xeon system with two 2.26GHz quad-core processors. The processors are based on the Intel Nehalem microarchitecture and have an 8MB L3 cache.

4.1 Computation-driven CPE for MiniSat

To evaluate the computation-driven approach to CPE we created variants of MiniSat's main solver routine in a very simple manner, using slightly different search constraints. Figure 3 shows an extract of the modified method *solve()* of class *Solver*. This method implements the restart policy of the SAT solver. It invokes the actual search routine with limits on the number of new boolean clauses that may be learnt during the search (*nof_learnts*) and the number of conflicts that may be encountered before the search routine aborts (*nof_conflicts*). Learnt clauses may reduce the search time by avoiding entering parts of the search space that do not lead to a solution, but they also potentially slow down the search. The selection of the upper bound for learnt clauses is therefore a trade-off between these two effects. If the search encounters too many conflicts, the solver restarts the search operation with an increased number of permitted conflicts and number of clauses that may be learnt.

In Figure 3, the code that was modified from the original program is emphasized in bold. Only two lines of code were added and one was changed to create a CPE-enabled program from the original sequential implementation. Line 4 initiates competitive execution with two variants. Variant-specific behavior is specified in line 5, where the second variant uses its variant-specific return value from `cpe_start()` to specify that an infinite number of clauses may be learnt during this search iteration. The other variant uses the default value of *nof_learnts*, which is increased after each unsuccessful search iteration. The while loop is executed on the order of 20–30 times for the data sets used. Sample runs showed that the execution time of the first few iterations is only on the order of a few milliseconds, and using competition is not attractive for these iterations. Competitive execution therefore only starts in the 5th iteration (not shown in the code), resulting in approximately 15–25 competitive phases.

---

[1] http://baldur.iti.uka.de/sat-race-2008
[2] http://www.satcompetition.org

```
Solver :: solve( nof_conflicts , nof_learnts ) {
  while (!done) {
     (...)
    v_learnts = cpe_start(nof_learnts, INF);
    done = search( nof_conflicts , v_learnts );
    cpe_finish();

     nof_conflicts  *= restart_inc ;
    nof_learnts     *= learntsize_inc ;

     (...)
  }
}
```

Figure 3: Program modification to enable CPE for MiniSat. Modified code is shown in bold.

```
 1  Solver :: solve( nof_conflicts , nof_learnts ) {
 2    mode = cpe_start(OUTER, INNER)
 3    while (!done) {
 4       (...)
 5      if (mode == INNER)
 6          v_learnts = cpe_start( nof_learnts , INF);
 7      done = search( nof_conflicts , v_learnts );
 8      if (mode == INNER)
 9          cpe_finish ();
10       (...)
11    }
12    cpe_finish();
13  }
```

Figure 5: Enabling nested CPE for MiniSat. Added code is shown in bold.

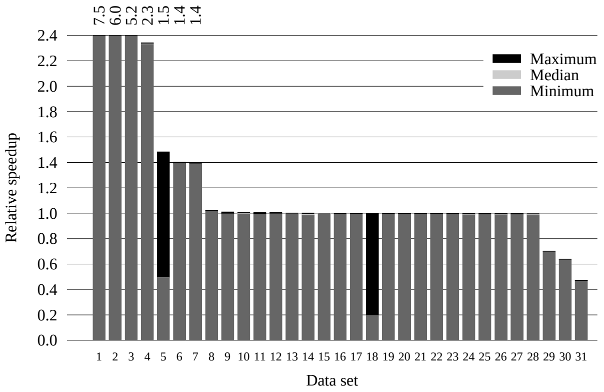

Figure 4: Speedup relative to a sequential execution for a CPE scenario using two variants.

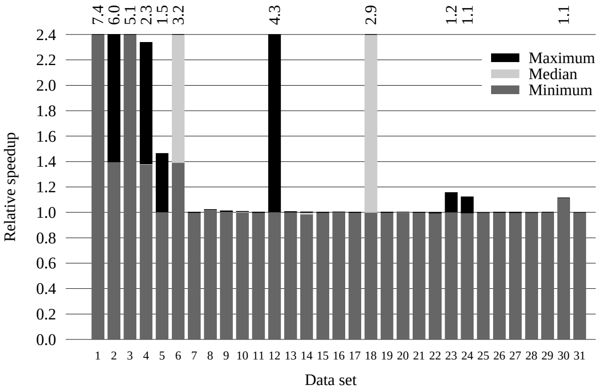

Figure 6: Speedup relative to a sequential execution for a nested CPE scenario with three variants.

4.2 Computation-driven CPE performance

Figure 4 shows the speedup that is obtained through this simple means of enabling CPE relative to a sequential execution of the program. Each measurement was performed three times. We report maximum, median, and minimum speedup for each of the 31 SAT benchmarks. The variation between the runs is negligible for most of the data sets, with two outliers. The data sets are numbered by decreasing maximum speedup achieved. The sequential baseline corresponds to a stand-alone execution of the first of the two variants. This configuration performs best for all input sets compared to other standalone variants and is therefore a fair base for comparison.

We first discuss the maximum speedups measured. For the first four data sets, the observed maximum speedup is super-linear with 7.5, 6.0, 5.2, and 2.3 times the sequential performance. For the three subsequent data sets, the speedup is sub-linear with 1.4–1.5 times the original performance. The speedups result from the second variant winning one or multiple competitive loop iterations. If the second variant wins an iteration, it has potentially pruned a larger part of the search space in less time than its competitor due to the larger memory for new learnt clauses. As a consequence, the total search time is reduced. The execution for data sets 29–31 is slowed down to 0.5–0.7 times the original performance. A slowdown can occur because the variant that wins an iteration determines the state of the solver and thereby the search path of subsequent iterations. In rare cases, this effect leads to an increased search time.

The median and minimum speedups are mostly identical to the maximum values. The two exceptions are data set 5 (slowdown to 0.5x) and data set 18 (slowdown to 0.2x). This variance in execution time between different runs stems from a very tight race between the competing variants in some loop iterations. For such iterations, the actual winner depends on external timing factors, and the program execution may therefore differ from one run to another.

4.3 Leveraging nested CPE

As we observe in the previous section, the introduction of competitiveness may in some cases result in a slowdown due to changes in the execution flow. Such changes can occur if the program state after a competitive phase depends on the variant that has won the phase. A measure to avoid possible negative performance impacts due to such effects is to run the original, unmodified version of the algorithm alongside the diversified variants.

The usage of nested competitiveness allows a program to do so in a straightforward manner. Figure 5 shows a possible way of modifying the program in Figure 3 to achieve this behavior. In line 2 execution is split into an outer variant that executes the original algorithm and an inner variant. The inner variant is (in line 6) in-turn split into two diversified variants that compete in every iteration of the while-loop. The program resumes to sequential execution as soon as the `cpe_finish()` call in line 12 is reached by either the variant that executes the original algorithm or the diversified variant that has won the last competitive loop iteration. At any point in time, at most three variants are executing in parallel.

Figure 6 shows the execution speedup obtained by the nested three-variant setting. The best-case performance tops the non-nested case with super-linear speedups for five data sets (7.4, 6.0, 5.1, 3.2, and 4.3) and sub-linear speedups for six other data sets (2.9, 2.3, 1.46, 1.16, 1.12, and 1.11). For the slowest of the three CPE runs, performance reaches 0.99 times the original performance in the two worst cases (data sets 14 and 22). For all other data sets, CPE performance reaches at least the original performance. For most of the data sets without performance improvement, the CPE-based execution with three variants corresponds to an actual sequential execution (i.e., all competitive phases are won by the unmodified variant). The absence of a noticeable slowdown for these data sets demonstrates that the overhead of competitive execution (induced by variant synchronization through the operating system's copy-on-write mechanisms) and the impact on shared resources such as the memory bus and shared caches are minimal.

The evaluation shows that using CPE, even a very simple program modification can lead to a remarkable performance improvement. Using a nested CPE scenario, the run-time system is able to provide the potential performance of CPE while guaranteeing that the lower performance bound comes close the performance of the original sequential program. More sophisticated forms of diversification and competitiveness can of course be introduced and may well lead to even better performance improvements. Another example on how CPE can be employed to speed-up heuristic search problems such as SAT is to partition the search space among different variants, e.g., by employing variant-specific randomization strategies to determine the order in which the variables of the boolean formula are assigned.

4.4 Compiler-driven CPE

The following four steps are used to create a CPE-enabled version of MiniSat using the compiler-driven approach.

1. Compiler optimization strategies that are beneficial for different input sets and run-time conditions of the program are used as input into the process. For the evaluation, we use the open source tool Acovea[3] to determine good optimization strategies for four randomly selected data sets. Acovea uses a genetic algorithm and an iterative compile/execute cycle to find a local optimum in the large space of possible optimization combinations. Table 1 shows the GCC optimization settings determined by Acovea for the four learning data sets.

2. The execution profile of the sequential program is analyzed to determine the program parts that contribute most to the program's execution time. Enabling CPE for these program parts potentially leads to the highest performance improvement. For MiniSat, this analysis reveals that more than 90% of the execution is spent in five methods.

Options	**Configurations**
-O1 -fcse-follow-jumps -finline-functions -fgcse -finline-small-functions	● ● ● ●
-fno-if-conversion -fstrict-aliasing -ftracer	● ● ● ○
-fno-guess-branch-probability -fpredictive-commoning -finline-limit=700 -fno-tree-sra -fno-tree-sink	● ● ○ ●
-fgcse-las -ftree-loop-im -ftree-vrp -fno-tree-copyrename	● ○ ● ●
-fno-merge-constants -fno-split-wide-types -foptimize-register-move	○ ● ● ●
-falign-jumps -fbranch-target-load-optimize -fmodulo-sched	● ● ○ ○
-fivopts -foptimize-sibling-calls -fpeephole2 -ftree-loop-ivcanon -funroll-all-loops	● ○ ● ○
-freorder-functions -fno-cprop-registers -freschedule-modulo-scheduled-loops	● ○ ○ ●
-falign-labels -freorder-blocks	○ ● ● ○
-ffloat-store -fthread-jumps	○ ● ○ ●
-fcrossjumping -fforward-propagate -fgcse-after-reload -fno-tree-fre -fpeel-loops -ftree-vectorize	○ ○ ● ●
-fno-tree-dominator-opts -fprefetch-loop-arrays	● ○ ○ ○
-ftree-store-ccp	○ ● ○ ○
-fno-tree-ch -fno-tree-ter -fschedule-insns2	○ ○ ● ○
-fcaller-saves -fdelete-null-pointer-checks -fgcse-sm -fregmove -funroll-loops	○ ○ ○ ●
-fno-if-conversion2 -fno-tree-ccp -fno-tree-copy-prop -fno-tree-dce -fno-tree-salias -fexpensive-optimizations -funswitch-loops -fbranch-target-load-optimize2 -fno-inline -fvariable-expansion-in-unroller	○ ○ ○ ○

Table 1: GCC optimization configurations determined by Acovea based on four training data input sets and used to generate the program variants. Each row aggregates the options enabled for a specific subset of the four distinct configurations.

3. A source-to-source transformation creates four clones of these methods and patches the method call-sites such that a variant-specific method clone is called instead of the original method. This behavior is achieved by using variant pointers (passed to and returned by `cpe_start()`) to index an array of method pointers. The original direct method calls are replaced by indirect calls through these method pointers.

4. The cloned methods are separately compiled and optimized using the optimization settings determined in the first step and shown in Table 1. All parts are finally linked into a single CPE-enabled program.

The steps to create a CPE-enabled executable were performed manually for this evaluation, but the whole process may easily be automated.

In the compiler-driven CPE scenario for MiniSat, the competition granularity is a single iteration of the main loop of the *solve()* method. The execution pattern of the CPE-enabled program is therefore similar to the computation-driven scenario presented in the previous section. Variants compete during one loop iteration. As soon as a variant completes an iteration the state of all variants is synchronized with this winning variant.

---

[3] http://www.coyotegulch.com/products/acovea

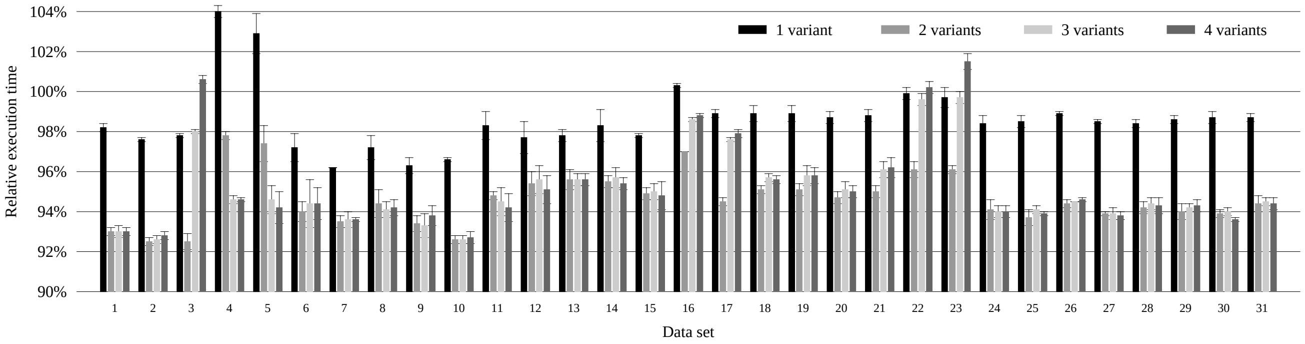

Figure 7: Execution time under CPE relative to the sequential program compiled with -O3. Numbers are shown for scenarios with one to four variants enabled.

4.5 Compiler-driven CPE performance

Figure 7 shows the execution time of the CPE-enabled program relative to the execution time of the sequential program compiled with a best default compiler optimization setting (*-O3*). All configurations were run three times and the figure shows the average execution time along with the standard deviation. Numbers are shown for configurations from a single variant up to all four variants that compete in each competitive phase.

For most data sets, execution using a single variant is approximately 1–2% faster than the reference sequential execution. For three data sets, single-variant execution is slightly slower than the reference (up to at most 4%). The two-variant CPE configuration performs better than the single-variant version for all data sets, for many of them in the order of 4–6%. The execution with three and four variants is beneficial for two data sets (4 and 5) compared to the two-variant configuration. For most other data sets the difference between 2-variant and 3/4-variant performance is negligible. For a few cases, the overhead induced by the higher number of variants results in a visible performance reduction.

Figure 8 shows how many competitive phases each variant has won in the two-variant scenario. The total number of competitive phases varies between 18 and 30, depending on the data set. In most cases, the first variant performs best in around ten phases, and the second variant in the remaining phases. A closer inspection reveals that the first variant is faster mostly in the initial and shorter phases, the second variant in the later and longer execution phases.

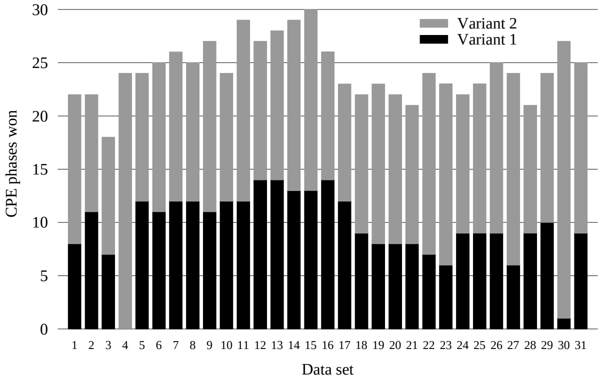

Figure 8: Count of competitive phases that each variant has won in the two-variant execution scenario. The numbers are reported for a single run of each data set.

4.6 Discussion

The evaluation results show that adapting sequential programs for competitive parallel execution leads to a performance improvement in many cases. The presented API consists of only three functions and enables a simple and straightforward modification of existing code. The isolation guarantees provided by the semantic model make reasoning about the behavior of CPE-enabled programs particularly easy. CPE simplifies the creation of programs that dynamically adapt to input data or other run-time characteristics.

Competitive execution of a program with multiple variants may, depending on the underlying architecture, of course increase the overall energy consumption compared to a single-variant sequential execution. It needs to be decided on a case-by-case basis if the benefits in improved program performance outweigh the increase in CPU usage and energy consumption for a given application scenario.

Good performance results can be achieved even with a purely software-based CPE-aware run-time system. Future hardware features may well make such an approach even more attractive by reducing the run-time overhead of memory isolation. Research in transactional memory systems [14], e.g., hint at the possibility that future computer systems may support more sophisticated memory isolation properties, e.g., hardware transactional memory (HTM). Depending on the actual semantics and performance that such HTM systems will provide, even more lightweight run-time systems to support CPE may partly rely on such transactional hardware support to provide memory effect isolation between variants.

5. RELATED WORK

A number of other research projects have investigated the usage of program variation at different levels to adapt programs to actual run-time conditions. Diniz and Rinard [8], e.g., investigate dynamic feedback in the context of a parallelizing compiler and dynamically select different code variants by alternating between sampling and production phases during execution. Code variants differ by the construct they use to synchronize execution among different threads. The ADAPT system by Voss and Eigenmann [22] dynamically compares and selects good candidates among different versions of hot loops. Versions of a loop are generated by compiling the loop code using different optimizations (e.g., using different levels of unrolling). Version generation is performed both offline and online during program execution, using a remote optimization system. Each version is executed at most once for a specific loop bound configuration, and the best version known at a given point in time is used in future invocations of the loop. Similarly, the online performance auditing system of Lau et al. [15] enables the online comparison of multiple compiled versions of code segments in a dynamically optimizing Java virtual machine. The system uses a statistical approach to compare different versions to enable comparison despite changing inputs and program state. Fursin et al.[11] present a method to compare the performance of multiple program versions in a single execution. In their approach, program variants are generated using different compiler optimizations, either offline for optimizations that change the code structure (e.g., the loop unroll factor), or online for optimizations that simply change a program variable (e.g., a parameter that specifies a loop tile size).

These approaches perform the selection of good variants in certain program execution phases, evaluating at most one variant at a time. The best known variant is then used for the remaining execution (or in separate program runs in the case of [11]). CPE, in contrast, does not require separate selection phases and competitively compares multiple versions at a time on multiple processors. A CPE-like runtime system with its capability for the concurrent evaluation of multiple variants and with its isolation properties may be leveraged to implement multi-versioning schemes like the ones discussed.

PetaBricks [1] is a parallel programming language with built-in support for algorithmic choice. The language provides constructs to specify multiple implementations for a problem. An automatic tuning system determines the actual algorithm configuration to be used for the specific execution environment. The tuning step takes place before the actual execution, and a single configuration is used at run-time. In CPE, variants can also be based on algorithmic variations, but instead of pre-selecting and running a single configuration, multiple configurations are executed in parallel and compete at run-time. Powerful language constructs like the ones included in PetaBricks may aid in the description of variants that are to be executed under a CPE model.

Yu et al. [25] present a framework for adaptive algorithm selection that chooses a suitable parallel algorithm from an existing library. The selection occurs dynamically and iteratively at run-time and is based on a characterization of the data input. The approach is specialized to parallel reduction algorithms, where input characterization based on a few parameters turns out to be successful for many studied benchmarks. In contrast, CPE is also applicable for scenarios where it is not possible to select the best performing algorithm based on a simple characterization of the data input.

Spiral [17] and Atlas [7] are two examples of automated library/code generators. Such tools aim at automatically generating and selecting the most efficient algorithm implementation for a given execution platform and problem set. Spiral and Atlas thereby target DSP transform algorithms and linear algebra programs, respectively. Such automated code generators can be used to generate variants to be executed under CPE. In cases where a single best implementation cannot be determined before program execution, a set of promising candidates can simply be executed using CPE.

Futures are a programming language construct that enables programmers to manually specify code blocks that may potentially be executed asynchronously and in parallel to other computations. Futures were initially proposed for MultiLisp [12], but have since been proposed for other languages, e.g., Java [23]. Similar to the Futures construct, the CPE model also provides a simple means to introduce code blocks to be executed in parallel, in the form of variants. In contrast to Futures, CPE variants execute in complete isolation from each other, and can therefore be run in parallel even if they perform conflicting memory operations.

In Orchestra [18], slightly modified variants of a program are executed in parallel on different processors to detect malicious intrusions, such as buffer overflow attacks. In PLR [19], the same program is executed multiple times in parallel to detect hardware faults. To detect intrusions or hardware faults, both approaches monitor the externally visible behavior of a program, which, in a correct execution, is identical in all program instances running in parallel. CPE employs techniques similar to Orchestra and PLR to orchestrate and monitor program execution, such as system call tracing. Unlike CPE, these approaches do not aim at improving performance of the running program.

Cho [4] describes an approach to exploit idle workstations in a network by competitively executing distributed applications as background processes. A similar technique is employed in MapReduce [6], where backup tasks of some work units are distributed to additional cluster nodes to deal with slow machines. Competitive parallel execution has conceptual similarities with these approaches but targets single multi-core systems rather than a set of networked machines. As a consequence CPE can leverage the tight coupling of cores to execute variants of loops or functions, in contrast to the use of remote workstations to execute identical copies of a program or task.

6. CONCLUDING REMARKS

Competitive parallel execution (CPE) is an attractive technique that allows inherently sequential programs to tap the parallel computing power offered by multi-core and future many-core systems. A CPE-enabled program contains several variants (for a loop, a function, or some other suitable program unit), and these variants compete at run-time. The fastest variant "wins" and terminates the other variants; as a result a program's execution time is the sum of the shortest variants. In this paper, we identify two approaches (computation-driven and compiler-driven CPE) and present a CPE-aware run-time system for the execution of CPE-enabled programs.

Computation-driven CPE employs variants that are identified in the program; often simple and localized modifications to an existing program suffice to create a CPE-enabled version of a program that competitively explores different alternative execution paths. *Compiler-driven* CPE exploits the fact that many optimizing compilers are unable to identify the best optimization settings for many programs (e.g., because input data determine a loop's trip count, or the size of the I-cache determines the best unroll factor). So compiler-driven CPE employs the compiler to generate different variants by selecting different compiler optimization strategies.

The CPE run-time system guarantees that competitively executed program parts run in full isolation and ensures that the CPE-enabled program has no side-effects that would diverge from a sequential execution. We present a simple low-overhead user-space library to provide these guarantees. We evaluate both approaches and demonstrate that the compiler-driven approach to CPE can result in a performance benefit even if the performance difference between differently optimized program variants is only on the order of a few percent. As an example for computation-driven CPE, we successfully modified a heuristic search algorithm that is part of a modern SAT-Solver. Even simple modifications to enable CPE result in super-linear speedups for some of the data sets evaluated.

As multi-core systems become more and more available, it is important to allow programs that are currently considered to be "sequential" to benefit from the parallel processing capabilities of the execution platform. CPE provides a simple approach that works even with today's architectures. If future architectures lower the cost of isolating the different variants, then the benefits will be increased. And if future programming languages capture more information on possible variants, then competitive parallel execution provides an attractive path to efficient execution on parallel systems.

7. REFERENCES

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. *PLDI '09*, pages 38—49, 2009.

[2] G. Bashkansky and Y. Yaari. Black box approach for selecting optimization options using budget-limited genetic algorithms. *SMART '07*, pages 1–16, 2007.

[3] 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, 2007.

[4] S. H. Cho. Competitive execution: a method to exploit idle workstations. *ICPADS '97*, pages 382–391, 1997.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7), Jul 1962.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI '04*, pages 137–150, 2004.

[7] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proc. of the IEEE*, 93(2):293 – 312, Feb 2005.

[8] P. Diniz and M. Rinard. Dynamic feedback: an effective technique for adaptive computing. *PLDI '97*, pages 71–84, 1997.

[9] N. Eén and N. Sörensson. An extensible SAT-solver. *SAT '03*, pages 502–518, 2003.

[10] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. *HiPEAC '07*, pages 245–260, 2007.

[11] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. *HiPEAC '05*, pages 29–46, 2005.

[12] R. H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[13] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability*, 6:245–262, 2009.

[14] J. Larus and R. Rajwar. *Transactional memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2007.

[15] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: using hot optimizations without getting burned. *PLDI '06*, pages 239–251, 2006.

[16] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. *CGO '06*, pages 319–332, 2006.

[17] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE*, 93(2):232 – 275, Feb 2005.

[18] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. *EuroSys '09*, pages 33–46, 2009.

[19] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Trans. Dependable and Secure Computing*, 6(2):135 – 148, Apr 2009.

[20] O. Trachsel, C. Fischlin, and T. R. Gross. A platform for competitive execution. In *ISCA Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA'08)*, pages 1–9, 2008.

[21] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. *CGO '03*, pages 204–215, 2003.

[22] M. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. *PPoPP '01*, pages 93–102, 2001.

[23] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. *OOPSLA '05*, pages 439–453, 2005.

[24] 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.

[25] H. Yu and L. Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, pages 1084–1096, Jan 2006.