

Supporting Application-Specific Speculation with Competitive Parallel Execution

Oliver Trachsel

Thomas R. Gross

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

Abstract

Parallel systems allow sequential programs that demand the highest possible performance or output quality to execute different versions of program parts in parallel to dynamically select the best version (i.e., the fastest or the one that produces the highest quality). The close coupling of multi-core systems offers new opportunities to explore such speculation. We discuss here how competitive parallel execution (CPE) supports such application-specific programmatic speculation. The key insight is that variations of the same program compete against each other during application-specific phases. These competing variants execute in complete isolation, thereby changing localized program state—comparable to a very coarse-grained transactional model. The state modifications of exactly one of these variants are committed and made globally visible based on an application-specific quality metric.

The paper discusses operating system and architectural features to support and further extend the applicability and versatility of application-specific programmatic speculation. It also motivates the need for more research on how future systems can accommodate the diverse requirements of speculative approaches at different abstraction levels.

1. Introduction

A software developer usually has many options on how to solve a problem at hand. Different algorithms, parameters, data structures, and implementations come with their respective properties

in terms of performance or output characteristics. Similarly, a user usually must configure a set of parameters when running a program. A good choice of parameters for both the programmer and the user (in terms of some situation-specific quality metric) often varies with the program's actual data input, and often cannot be determined without actually trying out different possibilities.

Application-specific programmatic speculation enables sequential programs to explore different execution paths simultaneously at run-time, taking advantage of the parallel processing power of modern multi-core and future many-core systems. The programs can thereby improve their performance or the quality of computed results. Suitable programming models and run-time support are required to enable many programmers to safely add such speculation to their programs. A central aspect to application-specific speculation models are isolation guarantees that relieve the programmer from worrying about possible side-effects between simultaneously explored execution paths. Such guarantees are especially important in the context of modern, highly complex programs, where a programmer is often not aware of the inner workings of the whole system, with all its components, underlying libraries, and the associated interactions. Due to this program complexity, manually modifying a program and using a traditional parallel programming approach to enable the exploration of alternative executions is a complex undertaking, that is often not considered worthwhile.

Competitive parallel execution (CPE) [18] is an instance in the design space of application-specific speculation models. In the CPE model, variations of the same program compete against each other during certain execution phases. Each execution path explored in parallel has a completely isolated view on the program state and all modifications are only visible locally to the specific execution path. At the end of each competitive execution phase, the resulting program state of one of the alternative execution paths is made globally visible and the program proceeds from this state. This effect isolation makes extending a program to use CPE particularly simple, as there is no need for complex reasoning on data sharing, mutual exclusion, data races, and other issues inherent to parallel programming.

We implemented a prototype CPE system to demonstrate that effective and efficient isolation guarantees can be provided by leveraging features of current operating systems. Additional features at the operating system and architecture level would enable even more powerful and versatile programming models and run-time systems, both in terms of performance and capabilities. CPE is a first step in the direction of supporting more general application-specific programmatic speculation, where programs can exploit idle processing power with minimal implementation complexity to pursue optional computations.

Depending on the underlying architecture and operating system, speculation may increase the energy consumption of a program. It needs to be decided on a case-by-case basis if the gain in terms of quality and performance outweighs this cost.

In earlier work [18] we have demonstrated how a subset of the competitive execution model and system can be leveraged to increase the performance of sequential programs. The present paper builds on top of that work and extends the CPE model to be applicable in a broader range of scenarios by giving the program more control over which of multiple speculative execution paths is committed and made globally visible. A similar concept, programmer-controlled competitive execution, has been described in [17] as an idea for future work. This idea has been adapted, implemented, and evaluated for the present paper.

2. Programmatic Speculation with CPE

The goal of application-specific programmatic speculation is to give programs a means to easily explore different alternate execution paths in scenarios where it is unclear which one of a set of algorithms, implementations, parameter sets, or heuristics is best suited to address a given problem. Application-specific speculation is not a general approach to parallelize applications, but rather fits for specific categories of applications, such as:

- *Algorithmic variants* – multiple algorithms or algorithm implementations exist and performance or quality characteristics depend on the input data. This group includes parametrized algorithms, heuristics-based algorithms, and randomized algorithms;
- *Transformation tasks* – data must be transformed from one representation into another, and some properties of the output depend on the configuration of the transformation method;
- *Search problems* – where the search space can be partitioned among different speculation units.

The CPE model needs strong isolation guarantees for speculative program variants to remove the need for an in-depth reasoning and understanding of issues that make traditional parallel programming intrinsically hard, such as data sharing, data races, and locking. Sections 2.1 and 2.2 describe the CPE programming and execution model in more depth, and Section 2.3 describes how features of modern operating systems and architectures can be leveraged to implement an efficient CPE run-time system. Modifying existing programs to enable application-specific speculation with CPE can be a very simple process, as illustrated by the video encoding example in Section 3.

2.1 CPE Programming Model

Our CPE API consists of three simple function calls to delimit the parts of a program where CPE is to take place:

```
void* cpe_start(int variants, void *v1, ...);  
void cpe_finish();  
void cpe_finish_wait(double quality);
```

These functions define the start and end points of competitive code sections. `cpe_start()` forks program execution into multiple competing execution flows. The call returns in multiple program variants, each one receiving the corresponding variant descriptor (vN) as return value. The variant descriptor can be an arbitrary, application-specific value, e.g., a pointer to a configuration for a parametrized algorithm or a pointer to a function to be called in the program variant.

Competitive execution takes place until *one* program variant invokes the second function call, `cpe_finish()`, or until *all* program variants invoke the third call, `cpe_finish_wait(quality)`. In the first case, competing program variants are aborted and the program proceeds from the state of the program variant that called the `cpe_finish()` function. In the second case, the completion of all competing program variants is awaited before the program proceeds from the state of the program variant with the highest quality value.

To enable CPE for an existing program it suffices to place these function calls at the appropriate places in the program and link the program with a CPE run-time library.

2.2 Execution Model

Upon forking program execution into multiple competing execution flows, the resulting program variants run in complete isolation from each other, and the effects and program state changes by one program variant are not visible in the other variants. These isolation semantics make reasoning about the behavior of variants particularly easy, as the program does not have to consider side effects of competitive execution versus sequential execution. If any of the program variants tries to perform an I/O operation that cannot be contained by the run-time system, competitive execution is prematurely aborted and program execution proceeds sequentially from the state of the program variant that performed the I/O operation.

A program can selectively relax these strict isolation semantics by allowing specific I/O operations (e.g., output to the console, reading or writing from/to specific files) and by explicitly sharing specific data among program variants. At the end of a competitive execution phase, only the program

state and effects of a single program variant remain visible to the program, except modifications to explicitly shared data and effects of explicitly enabled I/O operations. The CPE programming paradigm thus defaults to complete isolation of parallel entities (program variants) and allows for selective relaxation of this isolation. This behavior is in contrast to programming models based on the shared memory principle, where any effects of parallel entities (threads) are globally visible per default and special measures have to be taken to contain some of these effects (e.g., through the usage of thread-local variables or by protecting shared resources with locks). On the one hand, such a model where the default is complete isolation may have its restrictions and certainly has different applicability than a shared-memory model. On the other hand, the model makes it much easier to reason about the behavior and correctness of a program, because it requires explicit identification of resources that are not isolated from other speculative units and from the outside world.

2.3 Implementation

Our prototype CPE run-time system is a user-space library, but is closely bound to the operating system. (An earlier prototype of a system that in part provided similar functionality, and that was integrated into the operating system itself is presented in [17].) Program variants are represented as separate processes and the run-time system leverages the UNIX process model (where new processes are created as copies of existing processes) and the virtual memory management subsystem (which employs a copy-on-write approach to efficiently guarantee that memory modifications remain local to one program variant). The implementation thus uses processes rather than threads to obtain strongly isolated parallel entities. A similar path is taken, e.g., by the Grace system [2] to obtain safe concurrent programs.

The run-time system monitors I/O operations of competitive program variants through a system call inter-positioning mechanism. System calls with side effects that are not explicitly enabled by the program are intercepted and the program resumes single-variant sequential execution. System call interception is currently realized through the

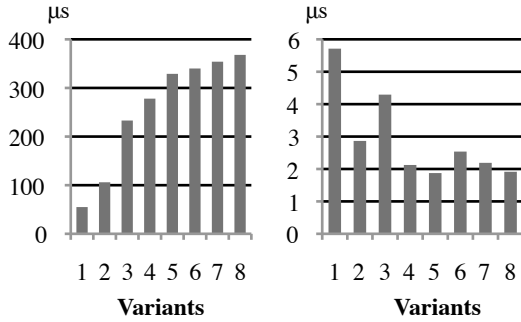


Figure 1. *Left:* Average time to start and terminate CPE variants. *Right:* Average time to intercept and inspect a system call.

operating system’s process tracing facility, which results in a context switch upon each system call by a program variant. Basic support by the operating system to perform a pre-filtering step and avoid a context switch for uncritical and explicitly enabled system calls would lead to performance benefits for system call intense applications.

More details on the implementation of the prototype run-time system are provided in [19].

2.4 Performance characteristics

This section gives a brief insight into the performance characteristics of the current run-time system prototype. The measurements presented in this section have been performed on an Intel Xeon system with two quad-core processors running at 2.26GHz. The processors (Intel E5520) are based on the Nehalem micro-architecture. Each core has a private L2 cache of 256 KB and the four cores on a processor share a L3 cache of 8 MB. The system has 12 GB of memory installed.

The left chart in Figure 1 shows the time it takes to create and terminate 1 to 8 CPE variants. This time varies between $55\mu s$ and $368\mu s$, depending on the number of competing variants. These times corresponds to an overhead of approximately $50\text{--}80\mu s$ per variant. The creation and termination overhead is constant and is therefore amortized over longer periods of competitive execution.

The right chart in Figure 1 shows the time required to intercept and inspect a system call invoked by a program variant. The time is measured by averaging the total time overhead to execute

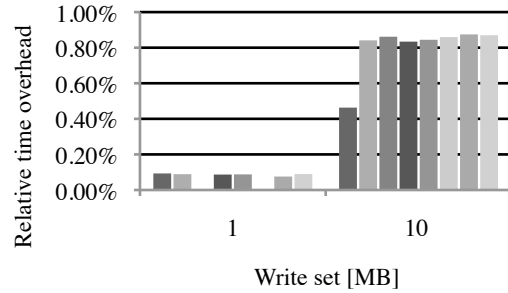


Figure 2. Execution time overhead due to copying of memory pages for 1 MB and 10 MB write sets per variant, for 1–8 variants.

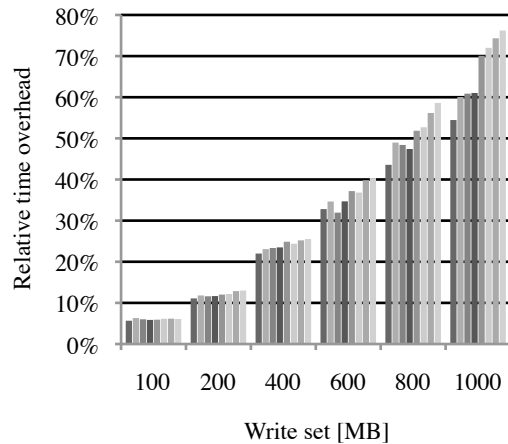


Figure 3. Execution time overhead due to copying of memory pages for 100 MB to 1000 MB write sets per variant, for 1–8 variants.

100,000 system calls per variant. The interception time varies between $5.7\mu s$ in the single-variant scenario and $1.9\mu s$ in the 8-variant scenario. The decrease in time overhead with increasing number of variants is a result of interleaving. This interleaving is possible because one part of the work to intercept a system call is performed on the CPU where the program variant is running, and another part on the CPU where the intercepting process is running.

Figures 2 and 3 show the execution time overhead due to copying of memory pages. Pages or copied to maintain variant-private program states. A variant-local copy of a memory page is created upon the first write to the page. The chart in Fig-

ure 2 shows the relative overhead if each of 1–8 variants writes to 1 MB or 10 MB worth of memory during a 1s period relative to a single-threaded non-CPE program that writes to the same amount of memory pages. The chart in Figure 3 shows the same data for write sets between 100MB and 1000MB. The overheads for a 1 MB write are below 0.01% for any number of variants. In the 10 MB case the overhead is always below 0.9%. For the bigger write sets, the execution time overheads are approximately 6% (100 MB), 11–13% (200MB), 22–26% (400MB), and 55–76% (1000MB).

These performance properties aid in deciding for a given scenario if application-level speculation with a CPE-like approach is practicable in terms of performance. Especially the memory behavior of an application should be taken into consideration. A CPE-like systems is well suited for applications with good locality properties with regards to modified memory locations. For such applications, the presented CPE run-time system prototype provides a low-overhead means to simultaneously explore multiple alternative execution paths.

3. Application Example

We have modified x264 [20], a modern H.264 video encoder, to illustrate the applicability and ease of use of a CPE-like approach for application-specific speculation, and to study the performance of the CPE run-time library. In this example, CPE is used to increase the picture quality of the encoded output video. Up to eight program variants, which use different encoding configurations, compete against each other for the encoding of each video frame. After encoding each frame, the program variant that produced the best picture quality relative to the original frame is selected and execution proceeds from this variant’s program state.

Figure 4 shows how the original program is modified to obtain a CPE-enabled version. Program variants are spawned in line 3, before the the actual encoding routine is invoked in line 4. Each program variant employs a different encoding configuration. Line 5 computes the quality of the resulting video frame, which is passed to the CPE completion routine invoked in line 6. The example shows that very simple code modifications can suffice to introduce

```

1  Encode_frame(pic_in) {
2  ...
3  cfg = cpe_start(nr_variants, cfg1, cfg2, ...);
4  x264_encoder_encode(cfg, pic_in, pic_out );
5  quality = x264_psnr();
6  cpe_finish_wait(quality);
7  ...
8  }
```

Figure 4. Program modification to enable CPE for the x.264 video encoder. Modified code is emphasized in bold.

multi-variant speculation for existing programs using the CPE model.

In x264, the encoding configuration is represented as an instance of type `x264_param_t`. Each variant can use its own configuration object. Variant-specific configurations can differ, e.g., in the settings that determine how a frame being encoded is analyzed. Such settings include the algorithm and parameters related to motion detection and estimation, various parameters related to psycho-visual optimizations, block partitioning settings, etc. These settings influence the quality and the bitrate of the resulting encoded frame. A suitable quality metric can then be used to select one of the resulting frames encoded by the different variants. Two examples of quality metrics that can be used are: i) to favor the encoded frame with the highest visual resemblance to the original image, computed in form of the peak signal-to-noise ratio (PSNR) or any other visual quality metric; or ii) to favor the frame with the smallest encoding size (i.e., the one minimizing the bitrate), but that fulfills some minimum visual quality requirements.

3.1 Speedup analysis

Figure 5 shows the speedup of the CPE-enabled program resulting from the modification shown in Figure 4 relative to a serial version. The serial version sequentially encodes each video frame multiple times in a row, once for each encoding configuration used by the CPE variants. An N-variant CPE-enabled execution competitively encodes each frame using N encoding configurations. The performance of this N-variant execution is compared to a serial version where each frame is

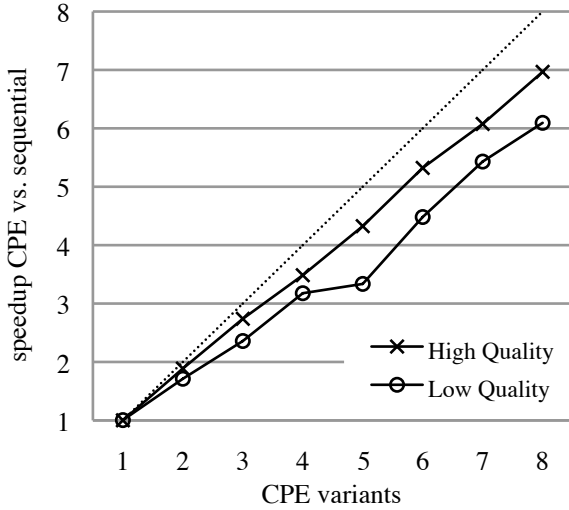


Figure 5. Speedup of the CPE-enabled version of x264 relative to a serial version that encodes each video frame multiple times in a row. The measurements were performed on an 8-core system based on the Nehalem micro-architecture.

encoded N times, one after the other, and using the same encoding configurations as the CPE-enabled version.

We have the program encode 100 frames, and use two different base configurations: i) *Low Quality*, a lower quality encoding where the encoding of one frame takes approximately 90ms on average, and ii) *High Quality*, a higher quality encoding taking approximately 500ms per frame. We are mostly interested in the performance and scalability of the approach. All CPE-variants therefore use the same encoding configuration to ensure that the stand-alone execution time of all variants is identical. The time to compute the resulting frame qualities is not taken into account in the reported data.

Figure 5 shows the speedup of the CPE-enabled program with 1–8 variants relative to the corresponding serial versions. The fine dashed line corresponds to a linear speedup. As expected, due to the overhead induced by the CPE isolation mechanics, the speedup of the CPE-enabled program is sub-linear. A regression analysis reveals that the speedup corresponds to approximately 0.72 times the number of variants for the low quality configuration, and 0.85 times the number of variants for the high quality configuration.

3.2 Overhead analysis

We compare the execution time of the CPE-enabled program with the time it takes to execute multiple identical instances of the original program in parallel. The goal of this comparison is to understand to which extent the overhead is inherent to the CPE approach and the current run-time system, and which part is due to the increased system load in general, such as contention on the memory-bus and shared caches. We perform this comparison on two 8-core systems that are based on different processor micro-architectures. The first system is based on the Intel Nehalem micro-architecture. The second system is based on the older generation Intel Core 2 micro-architecture. The remaining setup, including the encoder configurations, corresponds to the setup described in Section 3.1.

Figure 6 shows the execution times for the Nehalem-based system. Figure 7 shows the same data for the Core-based system. The two charts show the execution times of the CPE-based version (*CPE*) using 1–8 variants, and the execution times of a non-CPE version, where the same number (1–8) of identical encoding processes is run in parallel (*MP*, for multi-process).

For the CPE-based configurations on the Core-based system (Figure 7), we observe an increase in execution time from 9.0s to 17.9s (+99%) in the low quality configuration when increasing the number of program variants from 1 to 8. For the high quality configuration, the execution time increases from 41.5s to 60.0s (+45%). The corresponding increases in execution time for the multi-process scenario are 21% (from 9.0s to 10.9s) and 29% (from 41.5s to 53.6s), respectively. These numbers illustrate well that the execution times of the multi-process scenarios increase mainly with increasing contention on shared resources such as caches and the memory bus (the high quality encoding has a higher memory pressure than the low quality encoding, and lower spatial locality). The additional overhead for the CPE scenarios (which is mainly due to the process creation overhead and the copying of memory pages modified by program variants) decreases with an increasing computation/working set size ratio (higher quality encoding performs more computation, but does not access big-

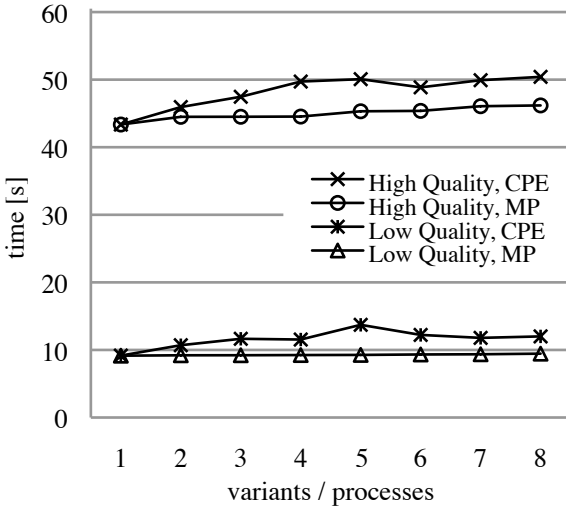


Figure 6. Performance of CPE-based video encoding on an 8-core system based on the Nehalem micro-architecture.

ger regions of memory than lower quality encoding).

The results for the Nehalem-based system in Figure 6 follow the same trends, but with a clearly lower relative increase in execution time for all scenarios. This better scaling behavior is due to the bigger caches and the higher memory bandwidth of this architecture. Execution time of the CPE-enabled program increases from 9.2s to 12.0s (+30%) for the low quality configuration, and from 43.4s to 50.4s (+16%) for the high quality configuration. In the multi-process scenarios the corresponding times increase by 3% (from 9.2s to 9.5s) and by 6% (from 43.4s to 46.2s).

In the low quality configuration (with its shorter speculative phases) the overhead that is due to the CPE-approach and run-time system themselves is thus approximately 27% (30%-3%). This overhead is significantly larger than the overhead observed for the high quality configuration (with the more than five times longer speculative phases) of approximately 10% (16%-6%). This difference is due to the fact that both configurations have comparable write set sizes for each encoded video frame and therefore a comparable amount of memory copying is required to maintain variant-local program states (approximately 3–10MB for most encoded frames). The time overheads due to this copying

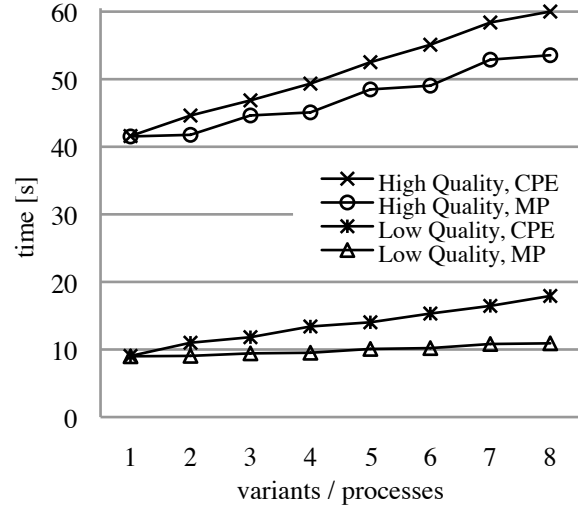


Figure 7. Performance of CPE-based video encoding on an 8-core Xeon system based on the Core micro-architecture.

and due to the creation and termination of variants can be better amortized over the longer speculative phases of the higher quality configuration than over the shorter phases of the low quality configuration. As a consequence, we observe a lower relative overhead in execution time.

An interesting artifact is the local peak at four variants in the CPE scenario for both the low and high quality cases. Further analysis has shown, that this behavior is partly influenced by the scheduling of variants to available cores. Different scheduling strategies lead to different trends. Additionally, the decrease in execution time for five and six program variants may also partly be due to synergetic effects. Similar effects have been observed and analyzed, e.g., by Zhuralev et al. [22].

3.3 Nehalem Turbo Boost

The measurements on the Nehalem-based system were performed with the processor’s Turbo Boost feature [9] enabled. This Turbo Boost technology allows processor cores to run at clock frequencies higher than the base operating frequency, and is activated and scaled depending on the number of active cores, estimated current and future power consumption, and processor temperature.

In theory, this boosting feature could lead to performance advantages for configurations with a

lower number of CPE variants or concurrent processes, compared to configurations with a higher number of processes. In practice, we have not observed such a bias in favor of single- or low-variant configurations. Compared to an execution with Turbo Boost disabled, the boosting feature leads to a speed-up in the order of 3–5%, independent of the number of CPE variants or concurrent processes; with the exception for 2 out of 32 configurations, where a speed-up of 1–2% is observed. This observation correlates with a study of the Turbo Boost feature by Charles et al. [3]. In their study, Turbo Boost leads to a reduction in execution time for all benchmarks and benchmark combinations. The benchmarks taken into account in the study represent a wide range of CPU-intensive and memory-intensive floating point and integer applications. Similar to our benchmarks, the authors did not observe a specific bias of Turbo Boost in favor of scenarios with a low number of used cores.

As a consequence, the availability of Turbo Boost on current processors needs not be taken into consideration upon deciding if a speculative model such as CPE should be employed or not for a concrete application scenario.

4. Beyond CPE

We see CPE and CPE-like programming models as a first step toward more general and versatile models of application-specific programmatic speculation. Such programming models could enable applications to easily pursue potentially beneficial computations alongside the main execution flow, without requiring sophisticated reasoning about possible side-effects.

The presented CPE run-time system can already offer acceptable performance for many application scenarios by leveraging features of modern operating systems and architectures. Additional features could enable CPE-like programmatic speculation models with even better performance characteristics and extended capabilities.

If, like in the presented CPE run-time system, the virtual memory system is leveraged to provide memory effect isolation among speculative program variants, the operating system’s copy-on-write implementation has a major impact on the overhead. This effect is also illustrated by the per-

formance measurements presented in Section 3, that indicate that the computation/write set size ratio has an influence on the performance behavior of CPE-enabled applications. This impact could be better adapted to the actual program and its memory access pattern by supporting heterogeneous memory page sizes. The memory copying overhead could thereby be reduced by selecting smaller page sizes for memory regions that contain small individual objects, and larger page sizes for memory regions that contain large objects that are usually modified as a whole (such as the encoded output frame in the video encoding example). Such support of heterogeneous memory page sizes requires modifications at the operating system level (virtual memory management) and at the architecture level (translation lookaside buffer). Heterogeneous memory page sizes could also benefit software transactional memory systems to implement cheaper and finer granular write barriers.

Transactional memory systems might in turn be leveraged for CPE-like execution models if they a) support a deferred update model and optimistic concurrency control with conflict detection and resolution taking place only at commit time; and b) provide a mechanism to disable automatic retry of aborted transactions. Transactions would be very long running in a CPE-based usage scenario, and a TM system would need to accommodate such characteristics in order to be applicable. CPE serves here as an example of applications of transactional memory systems that go beyond the original intention of such systems. We argue that such non-traditional usage scenarios should be taken into account when designing transactional memory systems.

To support application-specific speculation systems in managing speculative tasks, operating systems should offer more sophisticated interfaces to the scheduling subsystem than are currently available. As a minimum requirement, such interfaces would provide means to distinguish between compulsory and optional tasks and run optional (speculative) tasks only if and when resource and energy constraints allow to do so.

One major limitation of the current CPE system is that program variants cannot perform many types

of I/O operations without either prematurely aborting competitive execution and returning to a sequential single-variant execution, or changing externally observable state (by I/O operations that are explicitly enabled by the program). This limitation is a side-effect of the strong isolation guarantees provided by the CPE model. A possible solution to alleviate this limitation are operating system transactions, e.g., like [16]. Support for transactions in the OS would allow for more I/O operations inside speculatively executed program variants, by executing these operations in a transactional manner and committing only the operations of the finally selected program variant.

5. Related Work

Many other speculative techniques at different abstraction levels have been discussed in the literature. This section gives a brief overview on some of them and motivates that further research is required to address the question of how future computer systems can best accommodate and unify their diverse requirements.

One class of research investigates speculation at very fine granularities. Such low-level speculative techniques include (hardware, software, and hybrid) transactional memory systems, e.g., [6, 11], as well as hardware-supported thread-level speculation approaches, e.g., [8, 14].

Other research employs speculative techniques at a coarser level, spanning a larger part of a program's execution flow. Such approaches include, e.g., the Galois system for optimistic parallelism [12], the Fast Track system for speculative program optimization [10], and Master/Slave Speculative Parallelization [23], a hardware-based approach to speculative parallelization.

Approaches at an even coarser level make use of speculative techniques at the language-level, where speculation is guided by the program and incorporates even larger parts of a program. In terms of the abstraction level and the speculation granularity, these approaches come closest to the CPE approach. Warth and Kay [21] present a programming language construct that enables to fork computations of a program that operate on their own copy of the program state. The state modifications of such a *world* can later be committed back into

the main execution flow. A prototype of the system is implemented as an extension to JavaScript. Moura et al. [15] present an idea similar to CPE in the context of the logic programming language Logtalk. In their approach, the bodies of alternative clauses for the same goal are executed concurrently and compete to provide an answer. Cledat et al. [4] propose Opportunistic Computing, an approach to increase a program's performance or its realism under pre-defined responsiveness constraints by using spare computing cores to execute algorithm alternatives. In contrast to CPE, accesses to shared state need to be replaced by calls to an API to ensure that each algorithmic variant operates on a unique copy of all data.

Other research proposes frameworks, programming languages and language extensions to provide the means to specify algorithmic choice. Such systems can determine beneficial algorithm selection offline in some cases. For cases where offline selection is not possible, or where multiple algorithms or algorithm combination are potentially well suited, high level speculation support such as CPE can be used to simultaneously explore multiple possibilities at run-time. Examples of such systems are PetaBricks [1], the work of Li et al. to optimize sorting with genetic algorithms [13], the Selector language construct [5], and Du and Agrawal's system to support adaptive applications [7].

Generally, as the number of computing cores in systems continues to increase, it is to be expected that speculative approaches will become increasingly popular. Further research is needed to investigate how the diverse requirements of speculative techniques at various abstraction layers can best be unified and met by future computer systems. Another open research question that needs to be addressed is to what extent speculative and transactional approaches at different levels can co-exist or even be combined, and what kind of synergies and conflicts exist between the different techniques and their respective requirements.

6. Conclusions

The competitive parallel execution (*CPE*) model and run-time system show that application-specific multi-variant speculation is a viable possibility to leverage multi-core systems for different kinds

of applications and they can be efficiently implemented on current architectures and operating systems. Future systems could enable even better performance characteristics and extended capabilities for CPE-like speculative models by providing adequate support at the architecture and operating system level. Also, future research needs to investigate how speculation approaches at different abstraction levels can co-exist, and how future system can accommodate their diverse needs in a unified manner.

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] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. *OOPSLA '09*, pages 81–96, 2009.
- [3] J. Charles, P. Jassi, A. N. S, A. Sadat, and A. Fedorova. Evaluation of the Intel Core i7 Turbo Boost feature. *IEEE Intl. Symp. on Workload Characterization (IISWC '09)*, pages 188 – 197, 2009.
- [4] R. Cledat, T. Kumar, J. Sreeram, and S. Pande. Opportunistic computing: A new paradigm for scalable realism on many-cores. *First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, pages 1–6, Mar. 2009.
- [5] P. Diniz and B. Liu. Selector: A language construct for developing dynamic applications. *LCPC '02*, 2002.
- [6] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. *PLDI'09*, pages 155–165, June 2009.
- [7] W. Du and G. Agrawal. Language and compiler support for adaptive applications. *SC'04*, pages 1–12, Jan. 2004.
- [8] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *ASPLOS'98*, 1998.
- [9] Intel Corporation. Intel Turbo Boost technology in Intel Core microarchitecture (Nehalem) based processors. *Whitepaper*, pages 1–12, Nov 2008.
- [10] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. *CGO'09*, pages 157–168, Mar. 2009.
- [11] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. *POPL'10*, pages 1–12, Jan. 2010.
- [12] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *PLDI'07*, pages 211–222, 2007.
- [13] X. Li, M. Garzaran, and D. Padua. Optimizing sorting with genetic algorithms. *CGO'05*, pages 1–12, Mar. 2005.
- [14] 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, 2006.
- [15] P. Moura, R. Rocha, and S. Madeira. High level thread-based competitive or-parallelism in logtalk. *11th Intl. Symp. Practical Aspects of Declarative Languages, (PADL'09)*, pages 1–15, Jan. 2009.
- [16] D. Porter, O. Hofmann, C. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. *SOSP'09*, pages 161–176, Oct 2009.
- [17] O. Trachsel, C. Fischlin, and T. R. Gross. A platform for competitive execution. *ISCA Workshop on Parallel Execution of Sequential Programs on Multicore Architectures (PESPMA'08)*, pages 1–9, June 2008.
- [18] O. Trachsel and T. R. Gross. Variant-based competitive parallel execution of sequential programs. *Proc. ACM Intl. Conf. on Computing Frontiers (CF '10)*, pages 197–206, 2010.
- [19] O. Trachsel and T. R. Gross. Variant-based competitive parallel execution of sequential programs (extended version). Technical report 664, ETH Zurich, Laboratory for Software Technology, June 2010.
- [20] VideoLAN Project. x264 - a free h264/avc encoder. <http://www.videolan.org/x264.html>, 2010.
- [21] A. Warth and A. Kay. Worlds: Controlling the scope of side effects. VPRI Research Note RN-2008-001, Viewpoints Research Institute, Sept. 2008.
- [22] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ASPLOS'10*, pages 1–13, Mar. 2010.
- [23] C. Zilles and G. Sohi. Master/slave speculative parallelization. *MICRO'02*, pages 85–96, 2002.