SWAP: Parallelization through Algorithm Substitution

HENGJIE LI, WENTING HE, YANG CHEN

SKL Computer Architecture, ICT, CAS
Gradudate School, CAS, China
{lihengjie,hewenting,chenyang}@ict.ac.cn

LIEVEN EECKHOUT

Ghent University, Belgium
lieven.eeckhout@elis.UGent.be

OLIVIER TEMAM

INRIA, Saclay, France
olivier.temam@inria.fr

CHENGYONG WU

SKL Computer Architecture, ICT, CAS
cwu@ict.ac.cn

SWAP: Parallelization through Algorithm Substitution

Abstract—

The main impediment to a broad adoption of parallel programming is the need for programmers to grasp at least basic notions about parallel execution models.

In this article, we propose to involve the programmer in a different manner, more compatible with the knowledge of most programmers. Unlike parallel execution models, most programmers are reasonably familiar with the algorithms used in their program. By explicitly indicating which algorithms are used, and by encapsulating these algorithms within software components, programmers make it possible for an *algorithm-aware compiler* to replace their original algorithms implementations with compatible parallel implementations, or with the parallel implementations of compatible algorithms.

In order to capture the compatibility information, we introduce a novel compiler structure called the *Specification Compatibility Graph*, and a software environment for performing algorithm-aware compilation. We apply the approach to the parallelization of 5 benchmarks on four cores. Using 4 more benchmarks, we show that the same approach can take advantage of SIMD, or even of alternative algorithms for a given task. Overall, we achieve speedups ranging from 1.13× to 3.11×.

1 Introduction

Because automatic compiler-based parallelization is not yet mature enough to tackle a broad range of programs, in spite of significant recent advances [1], [2], parallelization is still likely to involve the programmer. As a result, the current key research issue is to make it as *easy* as possible for the programmer to parallelize applications. The most favored approach so far is to propose parallel environments which hide as much of the parallelization complexity to the user as possible. For instance, Intel TBB [3] or Cilk [4] facilitate parallelization by reducing it to the intuitive notion of splitting a task into sub-tasks, shielding the user from mapping, scheduling and other issues, all handled by the runtime environment. However, the user is still involved in the complex tasks of finding parallelism, and managing dependences.

Because the potential gains of parallelization are high, it is reasonable to expect that the user devotes some effort to transform the program. But most programmers are not proficient in parallelization techniques nor processor architectures; so the required transformations should remain in the realm of their expertise, and should involve as little architecture knowledge as possible.

There is already an approach that matches these characteristics: *parallel libraries*. In order to use parallel libraries, the only effort required from the user is to modify the program so that it matches the input/output interface of the library. At the cost of that transformation effort, any non-expert user can take advantage of the performance benefits of a parallel library. However, there are few parallel libraries, e.g., parallel STL — STAPL [5], MCSTL [6], thrust [7]. There is a vast set of tasks that programmers want to perform, and therefore few of the algorithms implemented in a given program are likely to be covered by existing parallel libraries.

We start from a simple postulate: while few programmers are proficient in architectures and parallelization techniques, there probably exists one or a few expert programmers capable of writing, or who even have already written, parallel versions of some of the key algorithms. Then, the key difficulty is to leverage that expertise and let non-expert programmers benefit from existing parallel versions of algorithms. The key concept behind SWAP is that non-expert programmers specify the semantic requirements of the algorithms in the program, and the SWAP framework then automatically swaps algorithms and implementations to optimize performance on a given processor architecture. By doing so, SWAP improves both performance and productivity.

To achieve this goal, we build upon three notions previously developed in the literature: (1) viewing programs as a *composition of algorithms* with glue code around them [8], (2) *community-based development* [9], i.e., the fact that programs are increasingly developed by bringing together code parts from many different sources (e.g., libraries, STL, PHP functions, or even code snippets found using Google Code Search), (3) the notion of *software components*, tightly encapsulated code parts with explicit communication interfaces, which are popular in software engineering [10], [11] because they allow several programmers to independently develop different code parts of a large program.

The principle of our approach is to make the compilation process *algorithm-aware* by writing (or rewriting) programs as compositions of algorithms wrapped into software components containing explicit algorithm information. The compiler parses the algorithm information and looks for *compatible* implementations in a software repository; then, the compiler instantiates the selected algorithm implementations and generates the final code. This repository is called by, but is *independent* from, the compiler. It can be independently populated by expert programmers, and thus, it constantly evolves and the newly added algorithm implementations are immediately available to programmers without any compiler and/or code modifications.

The key technical challenge in this approach is to determine whether two algorithm implementations can be substituted. This means (1) that the two algorithms must realize the same task, and (2) that their implementations (i.e., interface and platform requirements) are compatible. For that purpose, we propose the *Specification Compatibility Graph* (SCG), which captures all compatibility information within a single structure, such that it is *comprehensible for the compiler*. The SCG is the key enabler for the proposed SWAP framework which allows programmers to automatically substitute algorithm implementations. In contrast, traditional software libraries require the programmer to perform the exploration. The SWAP framework currently supports C/C++ programs: the programmer breaks up a program in different components and provides compatibility information using pragmas; the compiler does not require modification because the framework's syntax is processed by a pre-compiler. The framework then automatically searches for and substitutes alternative algorithms and implementations.

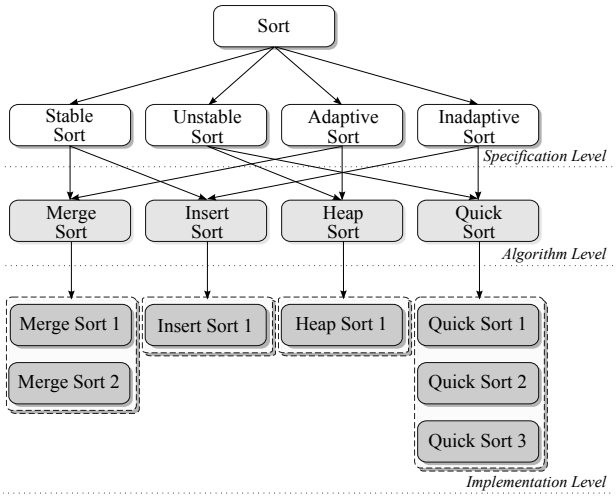

Fig. 1. *Specification Compatibility Graph (SCG).*

In the next sections, we explain how to capture the compatibility information of algorithms and their implementations, we illustrate the decomposition of programs into components, we briefly introduce the SWAP environment, and we provide some experimental evidence of both the low overhead and performance of our approach.

2 Specification Compatibility Graph

This section introduces the Specification Compatibility Graph (SCG) which captures the compatibility of algorithms and algorithm implementations.

2.1 Compatibility of Algorithms

We first explain how the SCG captures the compatibility of algorithms and how it depends on both qualitative and quantitative characteristics.

2.1.1 Qualitative Algorithm Characteristics

We distinguish three levels of abstraction for characterizing a program, see also Figure 1: the *specification*, i.e., the definition of what the program must be doing, such as sorting a list of data elements; the

algorithm, i.e., the method used to perform that task, such as Quick Sort or Merge Sort; and the *implementation*, i.e., programming that method, such as multiple different implementations of Quick Sort.

The specification is the only true constraint imposed by the programmer. In theory, any program abiding by that specification is compatible, independently of the algorithm and/or implementation used. For instance, once we know the target task is sorting, potentially, either Quick Sort or Merge Sort can be used. Thus, knowing the task specification opens up many alternative algorithms and implementation choices.

In practice though, just knowing the specification is not enough. It often happens that a programmer may need or want to use a specific category of algorithms, because they share certain properties for the task at hand. For instance, certain sorting algorithms are deemed Stable because they maintain the relative order of records with equal values. Therefore, the information should not just be Sorting, but Sorting and Stable/Unstable. At the same time, a user not concerned about the stability of a sorting algorithm can just specify sorting and use whatever algorithm is available. However, there is a myriad of such attributes for any task. For instance, sorting algorithms can also be Adaptive/Inadaptive (a sorting algorithm that takes advantage of the input order to speed up sorting), etc. One cannot expect to embed all such task-specific attributes for all possible tasks within a compiler — there are too many attributes and they evolve continuously.

However, a compiler does not need to know task-specific information, such as stable versus unstable sorting. It only needs to know whether an algorithm is compatible with a set of constraints specified by the programmer. One can note that algorithms, attributes and implementations can be organized into a *hierarchy*, which can be represented as the graph of Figure 1. In the graph, the nodes correspond to any type of abstraction information (task specification, algorithm, attribute, implementation), and the edges characterize the abstraction relationship ("more abstract than"). We call such a directed acyclic graph a *Specification Compatibility Graph* (SCG). This graph can deliver the compatibility information to a compiler in a task-independent manner. Using this graph, an algorithm is deemed compatible with a set of programmer-specified constraints simply if it belongs to the intersection of the spanning trees of all these constraints, whatever they correspond to. For instance, Merge Sort and Insert Sort are in the spanning tree of Stable Sort; but only Merge Sort belongs to the intersection of the spanning trees of Stable Sort and Adaptive Sort.

How is the SCG used by a programmer? In the code, the programmer specifies the graph node with the highest abstraction level compatible with the target task, as shown in Figure 2, and references the corresponding prototype in the program (we use a special prefix `cpt_` to distinguish component code from user code). Then, the compiler knows that it can use any instance located in the spanning tree of the specified node. Even though nodes are labeled with explicit abstraction information, e.g., Stable/Unstable Sort, this human-readable information is only provided for the programmer. The compiler only uses the relative relationship between the different nodes, not what each node stands for. The SWAP framework then automatically explores alternative compatible algorithm implementations and searches for the best one.

2.1.2 Quantitative Algorithm Characteristics

We need to distinguish between *qualitative* and *quantitative* characteristics of algorithms. Qualitative characteristics, such as sorting stability and adaptivity are expressed in the compatibility graph, as mentioned in the previous section. Quantitative characteristics cannot be expressed in the same way, due to the possibly infinitely large number of options. For instance, hashing algorithms, used for cryptography or compression, are characterized by their *strength*. A strength of 2^m means that a collision can occur between two hashed values after 2^m operations. (Operations are defined as bitwise operations such as *add, and, or, xor, rotate*; the number of such operations is used as the measure of the strength of a hashing algorithm.) Among the well-known hashing algorithms, for instance, $m = 25$ for MD5, $m = 39$ for SHA-0, $m = 52$ for SHA-1, etc. So, one could create SCG nodes corresponding to different intervals, e.g., $[20; 40]$ and $[40; 60]$, etc., and organize the algorithms accordingly. However, these intervals would be arbitrary, there is no clear

```
...
cpt_StableSort(...);
...
//cpt_pragma: hash strength >= 52
cpt_CryptographicHash (...);
...
```

Fig. 2. *Component reference and annotation for specifying a quantitative characteristic.*

way to break down the algorithms according to such quantitative characteristics.

Nevertheless, the programmer should have a way to restrict the algorithm selection based on these quantitative characteristics. For that purpose, when the programmer selects the abstraction level, we also provide the ability to specify quantitative characteristics using annotations, as shown in Figure 2, see `cpt_pragma`. These annotations are component-specific, and they are part of the component documentation available in the repository. Within the database, the existing component implementations optionally specify these quantitative metrics. If a programmer requests certain algorithm characteristics but the database contains no information about these characteristics for some of the algorithm implementations, they are deemed inappropriate for the target task, and are therefore ignored.

2.2 Compatibility of Implementations

Being able to express compatibility of algorithms is not sufficient, we also need the ability to express the compatibility of the algorithm implementations. The two essential aspects are the implementation interfaces and platform dependence.

2.2.1 Interfaces

Besides performing the same task (compatibility of algorithms), the algorithm implementation should use the same inputs and produce the same outputs (compatibility of implementations). This requires not just type checking of the inputs/outputs of the interface, but checking that the semantics (role of the inputs/outputs in the algorithms) are the same; again, such semantic checking is often beyond the reach of a compiler.

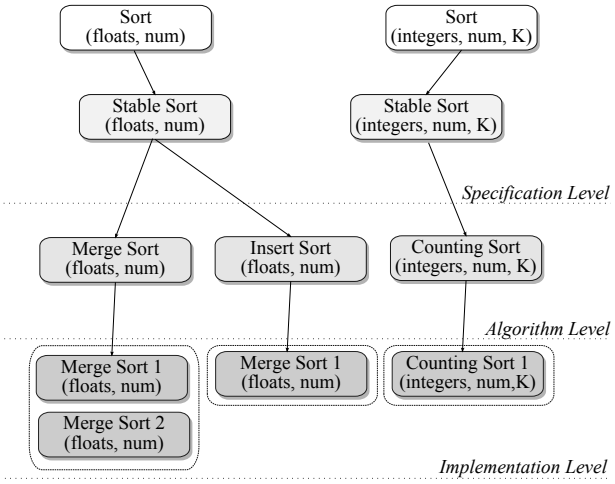

Fig. 3. *Compatibility of interfaces within SCG.*

There is no easy way for the programmer to express this information in a manner comprehensible by the compiler. This information is usually found in variable names and human-readable comments only. More sophisticated program specification techniques, such as UML [12], are meant to formally express such information, but they remain seldomly used by programmers.

However, it is possible to embed this information in the SCG so as to detect the compatibility of two implementations. As mentioned in Section 2.1, the SCG expresses implicit and relative abstraction relationships, as opposed to explicit and absolute abstraction information. We apply the same principle to the implementation interfaces. We never explicitly define the nature, type or data structure of the input and output parameters of the interface. We simply organize the implementations, within the graph, according to their compatibility. Consider the example of Figure 3. At any abstraction level, beyond the qualitative and quantitative algorithm characteristics, we are now also distinguishing nodes by their input/output interface. This includes both the data type information of each parameter, but also their *semantics*. For instance, Merge Sort, Insert Sort and Counting Sort are all stable sorts. Consider the case where the library

contains an implementation of Merge Sort and one of Insert Sort which both take as inputs a sequence of floating-point data and the number of data, and an implementation of Counting Sort which takes as inputs a sequence of unsigned integers and the maximum integer value. Then, there are two possible interfaces for stable sorts, each with a set of parameters with certain data type and semantics. As a result, we create two Stable Sort nodes, as shown in Figure 3.

2.2.2 Platforms

An algorithm implementation can be used on the target platform if the hardware (e.g., SIMD support) and software resources (e.g., TBB framework) it requires are available. As a result, much like compiler code generators require an architecture description file, our pre-compiler requires a *platform description file* that lists the available hardware and software resources; the database also records the hardware and software resources required by each component implementation.

3 Program Decomposition into Components

One of the key principles of our approach is that many programs can be decomposed into a set of components corresponding to various well-identified algorithms; we call this process *componentization*.

Componentization. The notion of *software components* is broadly used in software engineering [13]. However, our implementation of components is light and only remotely related to formal component frameworks. From software components, we essentially retain two key notions: *tight encapsulation* and *communications interfaces*.

Tight encapsulation means that the algorithm code contained in the component has no program side effect: it can only reference internal data or data provided in the communication interface. Furthermore, there is no naming conflict with the rest of the program (using either a special name suffix in C, or a dedicated namespace in C++; we use the former for now). Global variables cannot be used to communicate between components and the rest of the program, except if they are (redundantly) passed as parameters. The component takes the form of one or several C functions or C++ classes, except that the encapsulation contract is tighter than usual.

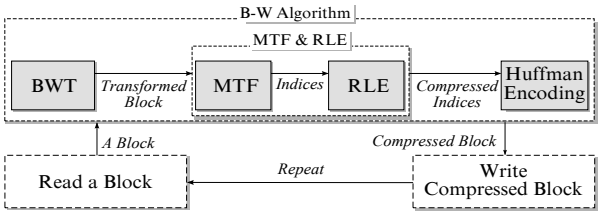

Fig. 4. *Block diagram of BZIP2; componentized algorithms are shown in gray.*

We illustrate program componentization with the SPEC CPU2006 version of BZIP2, see Figure 4, where each block corresponds to a component, and gray blocks correspond to components substituted by SWAP. The main modifications correspond to algorithm encapsulation and modification of the algorithm interfaces. For instance, we modified 25 lines to encapsulate MTF and RLE; 42 lines to adapt the interface of BWT; and 108 lines to adapt the interface of Huffman Encoding. The other benchmarks considered in this study are listed in Table1. The total number of source code lines that needed to be modified varies between 2 to 175, and less than 5.6% of the total program.

Parallelization. As explained earlier, sequential algorithm implementations can be automatically replaced by parallel implementations. Typically, parallelizing applications requires to carefully consider the interplays (locks, synchronizations, etc.) between the different program parts. However, this is not the case in the proposed approach for two reasons: first, by definition, the components are encapsulated, i.e., they do not modify global data structures, and second, they maintain the *sequential flow* of the components composing the program, i.e., components do not execute in parallel. As a result, the parallelization occurs *within* the component, where, on the other hand, expert programmers have implemented parallelism with the usual care for interplays between threads.

4 SWAP: An Algorithm Substitution Framework

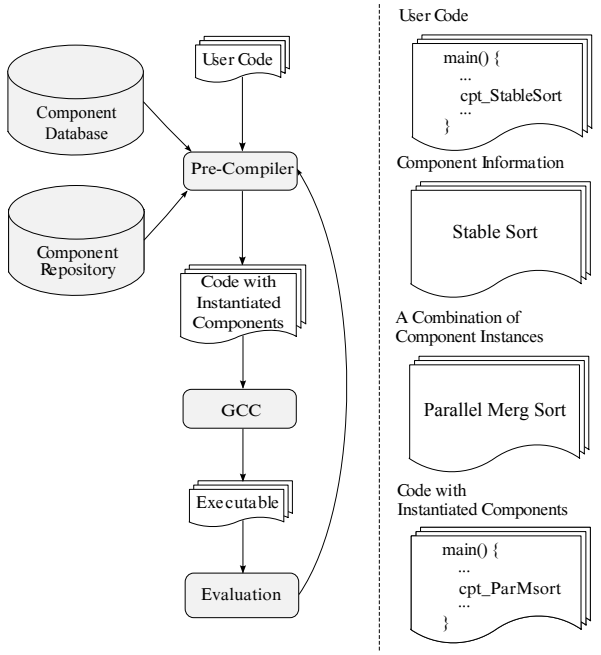

Fig. 5. *Framework and overall process.*

In this section, we briefly introduce the different elements of the SWAP framework. The overall framework is shown in Figure 5.

Pre-Compiler. Our current implementation targets C/C++ programs and components. Our method for referencing components and specifying annotations requires no special syntax, so the final program can be compiled using standard C/C++ compilers; we developed our pre-compiler as a front-end to GCC.

The task of the pre-compiler is four-fold, see Figure 5: (1) parsing the program in order to identify component references and annotations, (2) reading the SCG stored in the database in order to check for matching specification, algorithm or implementation nodes, (3) selecting the appropriate implementations (code and prototype) from the repository, (4) triggering the final compilation.

Database. The database consists of a set of tables which describe the SCG, essentially a *Nodes* and an *Edges* table. Table *Nodes* also indicates whether a node corresponds to an implementation in the software repository, or to a more abstract node such as *stable sort*, for instance.

Repository(ies). The repository is external to the pre-compiler in order to progressively augment it with new implementations or new algorithms as they become available, letting the programmer immediately benefit from them. Note that the pre-compiler can easily accommodate multiple repositories rather than just one.

In theory, for each newly uploaded algorithm implementation, the SCG maintainers must manually inspect the algorithm attributes (e.g., stable/unstable, adaptive/inadaptive for sorting, etc.), and the semantics of the interface parameters (e.g., set of data to sort, vectors or lists, etc.), and decide where it would fit within the SCG. In order to partially automate that process, we request limited information from the contributing expert programmer: to select among a progressively expanding list of algorithm *characteristics*, discussed in Section 2.1.1, and interface *semantics*, discussed in Section 2.2.1.

5 Experimental Results

After describing our experimental methodology, we evaluate the overhead of componentizing programs, the performance benefits of substituting algorithms and the impact of algorithm substitutions on algorithm-specific metrics.

Methodology. The benchmarks and their inputs are listed in Table 1. No particular criterion guided our selection, except for our relative familiarity with some of the algorithms used in the benchmarks. For DEDUP (PARSEC benchmark), we use the sequential version for the baseline performance, which is faster than the single-thread parallel version. Table 2 summarizes the componentization for all the benchmarks considered in this study. In Appendix A, we list the source of all alternative implementations in Table 2. We carried out our experiments on an Intel Xeon E5430 quad-core processor, with 12 MB L2 cache, and 16 GB of RAM. The compiler is GCC 4.4.2, and the OS is a version of RedHat Linux (4.1.2-44). For all benchmarks, the performance is averaged over 3

Benchmark	Domain	Cpt.	Lines	Mod.	%	Datasets
ASTAR (SPEC 2006)	Computer Games	1	5842	2	0.03	`Ref`
BZIP2 (SPEC 2006)	Data Compression	4	8293	175	2.11	`Ref`
CJPEG (MiBench)	Image Compression	1	26950	34	0.13	Gray16bit linear image
DEDUP (PARSEC)	Enterprise Storage	5	3683	121	3.29	`Simlarge`
LIBQUANTUM (SPEC 2006)	Quantum Computer	6	4357	157	3.60	`Ref`
VPR (vpr 5.0)	FPGA Placement	1	40197	141	0.35	`Stdev000`
BSDIFF	Linux Utility	2	608	34	5.59	Two versions of libruby-static.a
FREQMINE (PARSEC)	Data Mining	1	2710	41	1.51	`Simmedium`
SCOTCH	Graph Partitioning	3	12040	54	0.45	`Ncvxqp5`

TABLE 1

Benchmarks considered in this paper, along with the number of components (Cpt.), number of source code lines (Lines), the number of source code lines modified (Mod.), the fraction of source code modified during componentization.

Benchmark	Components	Algorithms/Implementations		#Combs
		Original	Alternative(s)	
ASTAR	Shortest Path Finding	Astar	Parallel bidirectional Astar	2
BZIP2	BWT	Seward's BWT	Div-BWT	12
	MTF	Move To Front	MTF-1 , MTF-2	
	RLE	Run Length Encoding	—	
	Entropy Encoding	Huffman	Arithmetic	
CJPEG	DCT	Seq-DCT	SIMD-DCT	2
DEDUP	Chunking	Rabin-Karp fingerprints	Fixed-size chunking	48
	Chunk Redundant Identification	Identifier	—	
	Crytographic Hash	SHA1	SHA, SHA256, MD5,SHA512, RIPEMD-160	
	Compression	Gzip	Bzip2, LZO, MGzip	
	Dynamic Sort	Heap sort	—	
LIB-QUAN-TUM	Pauli Spin Op1	Sequential	Parallel	64
	Pauli Spin Op2	Sequential	Parallel	
	Not Gate	Sequential	Parallel	
	Cnot Gate	Sequential	Parallel	
	CC-not Gate	Sequential	Parallel	
	Arbitrary 1-bit Op	Sequential	Parallel	
VPR	FPGA Placement	Simulated annealing	Deterministic parallel placement	2
BSDIFF	Suffix Sort	Qsufsort	Divsufsort	6
	Compression	Bzip2	Gzip, LZO, MGzip	
FREQMINE	FIMI	Fp-zhu	Afopt, Lcm	3
SCOTCH	Coarsen	Heavy-Edge match	First-fit match	12
	Initial Partition	FM	GPS , GGG	
	Refinement	Band-FM	FM	

TABLE 2

Components of each benchmark and variants for each component.

runs. We exhaustively explore all combinations of algorithm implementations, and report the best performance. All the algorithm implementations come from the public domain, they were either extracted from publicly available applications or libraries; we voluntarily did not re-implement any of the versions in order to highlight that there are multiple versions of important algorithms already available. Overall, our repository currently contains 51 different algorithm implementations, including multi-threaded implementations, as well as SIMD or alternative sequential implementations as explained later. The 51 algorithm implementations are organized into 23 different SCGs. The number of implementations in each SCG varies from 1 (which means we did not find compatible algorithms from the open-source domain) to 6.

Componentization Overhead. We first investigate the performance overhead of componentization. For that purpose, we compare the performance of the original (unmodified) sequential application against the componentized version. In this componentized version, no algorithm implementation has yet been substituted — the original program has simply been wrapped within components. As a result, the performance difference quantifies the overhead of componentizing a program (wrapping program parts within components). We define the overhead as the percentage increase in execution time of the componentized version over the original version. The measurements are presented in Figure 6. The highest overhead is 1.9% for BZIP2, and all other programs exhibit only negligible overhead, if not slight performance improvements. For DEDUP, the componentized version is actually faster because we modified the interface to pass the members of a struct directly to a function instead of the struct itself.

Substituting with Parallel Implementations. Using SWAP, we generated multi-threaded versions of 5 programs (ASTAR, BZIP2, DEDUP, LIBQUANTUM, and VPR) starting from sequential implemen-

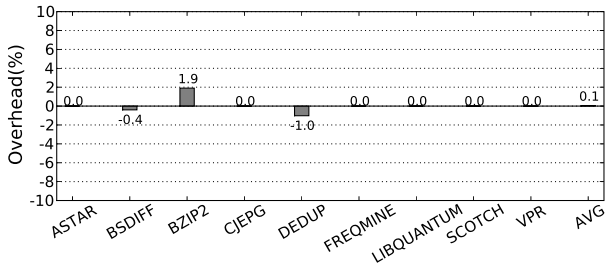

Fig. 6. Overhead of componentization over original version.

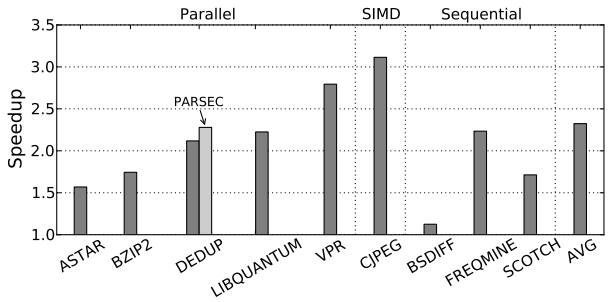

Fig. 7. *Performance improvement due to component substitution.*

tations. The speedups are computed over the *original* (unmodified) program version, and they range from 1.57× to 2.79×, see Figure 7. All programs use 4 threads, but the parallel ASTAR algorithm is designed for 2 threads only, because it parallelizes a path search by starting from both ends of a path. Quantitative specification is infrequently used in the cases we met. Only the security hash algorithm in DEDUP used it, which is depicted by Figure 2. It means programming with SCG is at least no more complex than calling a parallel library, in terms of code complexity.

SWAP versus manual parallelization (DEDUP). For DEDUP we did not use the parallel implementation provided in PARSEC, we started from the sequential version and substituted the original block compression with a parallel compression algorithm (MGzip). By varying the combinations of components, we created and evaluated 48 different implementations of DEDUP. The PARSEC parallel implementation resorts to pipeline parallelism, while we perform algorithm-based parallelization. In fact, in the PARSEC version, the compression stage is by far the most time-consuming, so there is one limiting stage in the whole execution pipeline. As a result, the benefit of application-based parallelization over algorithm-based parallelization is moderate, see Figure 7: the speedup is 2.12× (4 threads) for algorithm-based parallelization versus 2.28× for the application-based parallelization (see `PARSEC`). In general, application-based parallelization should outperform algorithm-based parallelization, but many programmers are just not capable of conducting application-based parallelization, while algorithm-based parallelization can be transparent.

6 More than Parallelization

While the initial goal of the proposed framework is to propose a parallelization approach for non-expert programmers, it also provides more opportunities for performance improvement than just parallelization. We illustrate these additional applications below but we leave their thorough investigation for further work.

Algorithm substitution. In some cases, just replacing a given algorithm with either, an alternative implementation of the same algorithm, or a compatible but different algorithm, can yield significant performance improvements as well. We illustrate this point in Figure 7, see "Sequential" substitutions, with 3 more benchmarks (BSDIFF, FREQMINE, and SCOTCH) listed in Table 1.

Hardware features. The same concept can be used to take advantage of special hardware features, such as SIMD, or accelerators, e.g., GPUs [14]. For instance, using SWAP, we substitute the sequential DCT implementation of CJPEG with a SIMD-based DCT implementation, and achieve a speedup of 3.11× as shown in Figure 7.

Algorithm-Specific Metrics. As mentioned in Section 2.1.2, programmers are also concerned with other metrics than performance. In some cases, they may want to bound these metrics so that algorithm substitution does not degrade them below a certain threshold, or they may accept to change them in order to further improve speed, or, on the contrary,

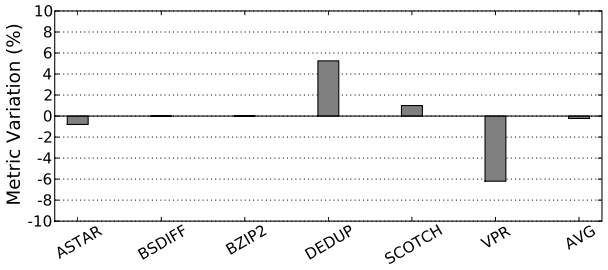

Fig. 8. *Impact of algorithms substitution on algorithm-specific metrics.*

they may want to optimize some of these metrics at the expense of speed. Among our benchmarks, ASTAR, BSDIFF, BZIP2, DEDUP, SCOTCH and VPR exhibit such tradeoffs. In Figure 8, we report the percentage of variation of each application-specific metric (i.e., average path length for ASTAR, patch size for BSDIFF, compression ratio for BZIP2 and DEDUP, graph cut size for SCOTCH, bounding box size and critical path delay for VPR) for the speedup results of Figure 7. The y-axis corresponds to the percentage of variation of the application-specific metric with respect to the original implementation. Positive metric variation means the framework can improve such metric after swapping to get the performance improved. For instance, in DEDUP, although the speedup of algorithm-based parallelization (by SWAP) is a litte lower than application-based parallelization (original PARSEC parallelization), see Figure 7, the compression ratio is improved by 5.3%, see Figure 8. For VPR, we can get a speedup of 2.79×, if we can tolerate 6.3% degradation on critical path delay. Except for DEDUP and VPR, the absolute variations are less than 1%.

7 Related Work

The notion of software components is broadly used in software engineering for allowing developers to work independently on large projects [13], and seamlessly modify and replace program parts. Component frameworks, such as JavaBeans [10], .Net [11] or the Fractal Open Component Model [15] usually define and enforce programming contracts, such as strict encapsulation of both code and data, explicit input and output interfaces, etc. Here, we mainly retain the notions of encapsulation and explicit interfaces.

To a certain extent some parallel environments implicitly or explicitly leverage the notion of components. For instance, both TBB [3] and Cilk [4] implicitly encapsulate tasks to be parallelized. Charm++ [16] explicitly and strictly encapsulates tasks in component-like structures because they are meant to be executed on a distributed-memory machine and to communicate via message passing. Linderman et al. [17] also propose to view programs as a set of individual components, and to efficiently map them on multiple cores. However, the parallelization is again done by the programmer using a MapReduce-like approach.

Thomas et al. [18] propose to encapsulate tasks such as sorting and matrix-multiply into the software containers provided by STAPL [5] in order to explore the best parallel versions of these algorithms. The STAPL framework embodies a number of compatible algorithms and features a prediction model constructed through machine learning to select algorithms. Pan et al. [19] want to facilitate the usage of the different existing parallel libraries, in order to help develop parallel applications. For that purpose, they propose low-level primitives and common interfaces for developers to use multiple different parallel libraries within their applications.

Like Petabricks [8], we promote an algorithmic-centric approach to programming. However, in Petabricks, these versions are written and embedded within the program itself by the programmer. One of our fundamental hypotheses is that most programmers are either unwilling or unable to write efficient parallel programs. Since, in Petabricks, alternative implementations are not sought externally, there is also no attempt at characterizing the compatibility of algorithms and implementations.

8 Conclusions and Future Work

We propose an approach to parallelization compatible with the skills of most programmers. This approach is based on feeding the knowledge of expert programmers, capable of parallelizing applications, to non-expert programmers.

In order to allow, and to maximize the likelihood of, automatic replacements of algorithms with parallel implementations, we introduce the SCG, a structure capable of capturing the compatibility information of algorithms and their implementations, in a manner comprehensible by a compiler. We present SWAP, a software environment which leverages this information, and we successfully apply it to the parallelization of five applications, and the overall optimization of nine applications.

Beyond the benefits for parallelization, we plan to extend the concept of substituting algorithms to supporting hardware accelerators in heterogeneous multi-cores: knowing which algorithm the program is performing allows to map the corresponding program section to a hardware accelerator (ASIC, FPGA or GPU) if it can support that algorithm.

REFERENCES

[1] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable Speculative Parallelization on Commodity Clusters. In *International Symposium on Microarchitecture*, pages 1–22, Atlanta, GA, 2010.

[2] Mohammed W. Benabderrahmane, Louis Noel Pouchet, Albert Cohen, and Cedric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303, Paphos, 2010. Springer.

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

[4] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, pages 207–216, August 1995.

[5] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In *LCPC*, pages 193–208, August 2001.

[6] Felix Putze. Johannes Singler, Peter Sanders. The Multi-Core Standard Template Library. In *Euro-Par*, 2007.

[7] Jared Hoberock (NVIDIA) Nathan Bell. thrust. http://code.google.com/p/thrust/.

[8] J. Ansel, C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks : A Language and Compiler for Algorithmic Choice. In *PLDI*, pages 38 –49, June 2009.

[9] S. Sethumadhavan, N. Arora, R. Ganapathi, J. Demme, and G. Kaiser. COMPASS: A Community-driven Parallelization Advisor for Sequential Software. In *International Workshop on Multicore Software Engineering (IWMSE)*, pages 41–48. IEEE Computer Society, May 2009.

[10] L. DeMichiel. *Enterprise JavaBeans Specification, Version 2.1*. Sun Microsystems, November 2003.

[11] A. Rasche and A. Polze. Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET. In *ISORC*, page 164, May 2003.

[12] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software*. Addison-Wesley Longman, October 2000.

[13] K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33:709–724, October 2007.

[14] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31:7–17, 2011.

[15] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The FRACTAL Component Model and Its Support in Java. *Software: Practice and Experience*, 36:1257–1284, September 2006.

[16] L. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA*, pages 91–108, October 1993.

[17] M. Linderman, J. Collins, H Wang, and T. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *ASPLOS*, pages 287 –296, March 2008.

[18] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. Amato, and L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *PPoPP*, pages 277–288, May 2005.

[19] H. Pan, B. Hindman, and K. Asanović. Composing Parallel Software Efficiently with Lithe. In *PLDI*, pages 376 –387, May 2010.