COMPASS: A Community-driven Parallelization Advisor for Sequential Software

Simha Sethumadhavan
Computer Architecture Lab

Gail E. Kaiser
Programming Systems Lab

Department of Computer Science, Columbia University, New York, 10027
E-mail: {simha,kaiser}@cs.columbia.edu

Abstract

The widespread adoption of multicores has renewed the emphasis on the use of parallelism to improve performance. The present and growing diversity in hardware architectures and software environments, however, continues to pose difficulties in the effective use of parallelism thus delaying a quick and smooth transition to the concurrency era. In this paper, we describe the research being conducted at Columbia University on a system called **COMPASS** *that aims to simplify this transition by providing advice to programmers while they reengineer their code for parallelism. The advice proffered to the programmer is based on the wisdom collected from programmers who have already parallelized some similar code. The utility of* **COMPASS** *rests, not only on its ability to collect the wisdom unintrusively but also on its ability to* automatically *seek, find and synthesize this wisdom into advice that is tailored to the task at hand, i.e., the code the user is considering parallelizing and the environment in which the optimized program is planned to execute.* **COMPASS** *provides a platform and an extensible framework for sharing human expertise about code parallelization – widely, and on diverse hardware and software. By leveraging the "wisdom of crowds" model [26], which has been conjectured to scale exponentially and which has successfully worked for wikis,* **COMPASS** *aims to enable* rapid *propagation of knowledge about code parallelization in the context of the actual parallelization reengineering, and thus continue to extend the benefits of Moore's law scaling to science and society.*

ACM Keywords: C.1.4: Parallel architectures **D.3.4:** Software optimization **F.3.2:** Program analysis **H.3.4:** Recommender systems **I.2.6:** Knowledge Acquisition **K.4.3:** Computer-supported Collaborative Work

1 Introduction

The adoption of chip multiprocessors (CMPs) poses methodological and linguistic challenges for sequential software. While new programming models and languages can help us create correct parallel programs quickly, there is an immediate need for tools that can systematically help in parallelizing, debugging and performance engineering of the vast sequential legacy code base. In this paper, we outline our vision for such a tool. **COMPASS** – A Community-driven Parallelization Advisor for Sequential Software – proffers advice to programmers based on information collected from observing a community of programmers parallelize their code. The utility of **COMPASS** rests in part on the premise that the growing popularity of CMP systems will encourage expert programmers to parallelize some of the existing sequential software, and **COMPASS** can quickly deploy capabilities to capture their wisdom (including any gained through trial and error intermediate steps) for the multicore software engineering community.

COMPASS observes expert programmers (henceforth called gurus) parallelize their sequential code using parallel programming patterns and other techniques for parallelization, records their code changes (before and after), summarizes this information and stores it in a centralized Internet-accessible database. When a relatively inexperienced new user (henceforth called a learner) wants to parallelize his/her code, the system first identifies the regions of code most warranting performance improvement (determined by profiling typical executions), and then which of those regions are most amenable to parallelization (by consulting its database of previously parallelized code). **COMPASS** then presents a stylized template, or "sketch", that can be used as a starting point for parallelization by the learner.

The learner may then provide feedback to the system on the usefulness of the advice. To effectively provide these capabilities, COMPASS builds upon recent work on code clone detection and graph matching algorithms, and introduces program transformations for generating program sketches from similar code.

The work described in this paper is still in progress but our pilot system can already analyze enterprise level code in C/C++. Once complete, COMPASS will provide a wide range of advice for improving application performance across multiple granularities of parallelism. Its practical utility will depend on the nature and number of users in the system and the diversity of their code. We believe that it is not unreasonable to imagine that COMPASS will be able to provide advice for a large class of present and upcoming CMP systems with templates covering: peephole Data Level Parallelism optimizations such as using SIMD kernels for inner loops, splitting loop iterations into threads using OpenMP, replacement and threading of large chunks of serial code with vendor-supplied optimized libraries (such as Nvidia CUDA), stream dataflow graphs, and thread and lock creation for irregular applications.

To the best of our knowledge, we are the first to propose leveraging collective human wisdom to propagate "best practices" about how to parallelize code. This is a new way of thinking about (multicore) performance engineering enabled by the connectedness brought about the Internet and the wide availability of parallel machines. COMPASS is a significant improvement over the state of the art, where the learner has to "pull" parallelization advice from books, class notes and/or Internet tutorial examples. Such a process can be time consuming and error prone. COMPASS, on the other hand, is a "push" based system where advice is proactively proferred to the learner with very little overhead for the learner. In addition, unlike tutorials, the advice is customized to the learner's specific coding problem. Further, unlike traditional hardware, compiler, language or hybrid approaches for parallelization, which have long incubation times before application end users can benefit, typically years to decades, COMPASS can enable rapid parallelization of sequential code because the usefulness of knowledge networks like COMPASS scale exponentially as the number of users in the system increases [20].

2 System Architecture

COMPASS targets two main groups of human actors: the gurus, who are experienced with parallel optimizations, and the learners, who are attempting to optimize their code for multi-/many-core machines, perhaps for the first time. Its a continuum: The same individual may operate as both guru and learner, presumably in different contexts and/or as expertise builds over time. We assume some initial set of (relative) gurus, rather than all learners. The purpose of COMPASS is to provide a framework and a set of mechanisms that makes it simple for the gurus to communicate their wisdom to the learners. One key aspect of the system is that the wisdom to be offered to learners is gathered from *multiple* gurus, some of whom may not be trustworthy nor their code optimal – then mined, customized and served with very little effort by either gurus or learners.

The architecture of COMPASS is shown in Figure 1. The system has four modules that combine to provide the required functionality – the watcher, the data store, the matcher, and the generator. The watcher observes how gurus parallelize their code by tracking the differences in the source code before and after an optimization. When an optimization is complete, the watcher snips out the code regions corresponding to the optimization, and summarizes the before and after versions in the form of unique identifiers, called *signatures*, and deposits them in the data store. The data store holds the before and after signatures as $< key, value >$ pairs (with the before version acting as the key and the after version serving as a value – which could in principle be reversed to de-parallelize if warranted). When a learner wants to parallelize some code, the matcher module prepares signatures of regions of code that are considered critical for parallelization (e.g., via hotspot profiling), and sends them to the data store. The data store runs a query with the matcher signature as a key, ranks the results and returns the top parallelized signature(s) to the generator. The generator then prepares a "sketch" of the code corresponding to the parallelized version, presented as a graphical overlay that can be accepted as is, modified or rejected by the learner. The learner may optionally provide feedback to the data store on the usefulness of the sketch.

§ **An Use Case** We envision COMPASS being useful in a wide range of optimization scenarios, only one of which we elaborate (refer to Sethumadhavan and Kaiser [21] for more

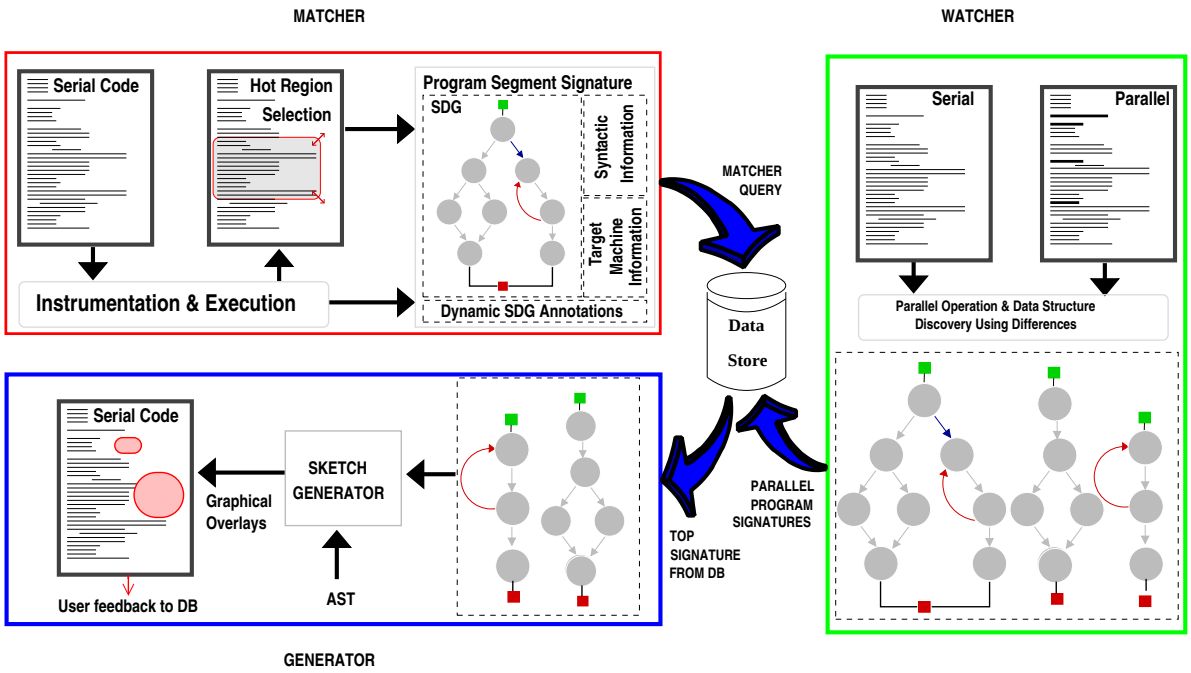

Figure 1. System architecture of the proposed COMPASS system.

examples). The example described below is chosen to highlight parallelization opportunities that **COMPASS** is likely to be able to exploit whereas typical optimizers – compilers and hardware – do not exploit.

§ **Procedurization** When chip companies design new hardware features to improve performance, they typically release APIs that can fully utilize the new hardware capabilities. Examples include the CUDA graphics library for effectively using the capabilities of NVIDIA Graphics Co-Processors [17] and the Intel Performance Primitives [25] which contain specially optimized library routines for domain-specific operations such as JPEG image processing or video transcoding using the SSE instruction extensions. To improve performance using these libraries without **COMPASS**, *each* code owner has to know about the existence of the optimized API, know how to use the API correctly, and then carefully adapt the code to invoke the API. On the other hand, with **COMPASS** advice, the developer is automatically informed of new/modified APIs and gets suggestions on code segments that can be proceduralized as soon as a few gurus have committed their corresponding code changes. An example of such an replacement is shown in Figures 2, 3. The LHS code listing of Fig. 2 shows a routine from an open source library (libJPEG), replaced on the RHS by an equivalent function call from the Intel Performance Primitives library. The LHS code listing of Fig. 3 shows a saxpy routine replaced on the RHS by an equivalent function call from the CUDA library.

While this is only one example optimization, in general, **COMPASS** can support optimizations covering both control and data changes, and consider different program granularities (peephole to global). We believe that over time, the number and sophistication of optimizations that **COMPASS** can support will grow, as more gurus join the system, and will ultimately be limited only by human ingenuity.

3 COMPASS Internals

Three important technical factors will determine the success of the **COMPASS** system: (α) effectiveness of signatures in uniquely representing program segments (matcher and watcher modules); (β) selectivity of optimization ranking procedures (data store module); and (γ) efficacy of algorithms used to reconstruct code sketches from signatures (generator module). We discuss traditional solutions to the above problems, point out their drawbacks, and describe the novel alternatives we are developing.

The power to handle large, multiple code bases – which are out of reach of most traditional techniques – comes from two key aspects of the **COMPASS** system: First, unlike tra-

Figure 2. Example COMPASS Procedurization using a library call from the Intel IPP®library [25]

```
void RGBToYCbCr_JPEG_8u_C3P3R(Ipp8u* pSrcRGB,
        int srcStep, Ipp8u* pDstYCbCr[3],
        int dstStep, IppiSize roiSize)
{
    for(int i = 0; i < roiSize.height; i++)

        for(int j=0; j < roiSize.width; j++)
        {
            int index = i * roiSize.width + j * 3;
            unsigned char R = pSrcRGB[ index];
            unsigned char G = pSrcRGB[ index + 1];
            unsigned char B = pSrcRGB[ index + 2];

            pDstYCbCr[0][i * roiSize.width + j]
             = 0.299*R + 0.587*G + 0.114*B;
            pDstYCbCr[1][i * roiSize.width + j]
             = -0.16874*R - 0.33126*G + 0.5*B + 128;
            pDstYCbCr[2][i * roiSize.width + j]
             = 0.5*R - 0.41869*G - 0.08131*B + 128;
        }
} /*Before */
```

```
void RGBToYCbCr_JPEG_8u_C3P3R(Ipp8u* pSrcRGB,
        int srcStep, Ipp8u* pDstYCbCr[3],
        int dstStep, IppiSize roiSize)
{
    #ifdef COMPASS
     ippiRGBToYCbCr_JPEG_8u_C3P3R(pSrcRGB,
     srcStep, pDstYCbCr, dstStep, roiSize);
    #else
       // original code
    #endif

/* After */
```

```
/*Before*/
// y = ax + y
void saxpy (int n, float a, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = a*x[i] + y[i];
}

//invocation
saxpy(n, k, x, y);
```

```
/*After*/
#ifdef COMPASS
__global__ saxpy_cuda(int n, float a, float *x, float *y) {
  int i = blockIdx*blockDim.x + threadIdx.x;
  if (i<n) y[i] = a*x[i] + y[i];
}

// invocation
int nblocks = ((n + 255) << 8);
saxpy_cuda<<<nblocks,256>>>(n,k,x,y);
#else
 // original code
#endif
```

Figure 3. Example COMPASS Procedurization using a library call from the NVIDIA CUDA®library [17]

ditional systems that require complete mechanization of the analysis process, **COMPASS** acknowledges the usefulness of human involvement (after all, programming is a human activity) and does not have to produce the best advice all the time. In the rare cases when **COMPASS** may produce bad advice (e.g., advice that is apparently irrelevant to the problem at hand), such as during the early database buildup stage, the programmer is free to discard the advice and ask for any known alternatives. Second, and perhaps more importantly, **COMPASS** mitigates scalability concerns by reducing the size of the code regions to be analyzed. It is widely believed that 90% of program execution time for most applications is spent in 10% of the code. **COMPASS** increases the effective scalability of traditional algorithms by focusing only on the most critical regions (that 10% of the code – profiled hotspots or marked by the user).

3.1 Program Representation

Conceptually, the **COMPASS** modules that use signatures to store and match parallelization solutions (derived parallel patterns) are similar in spirit to work on code clone detection – with two major distinctions: **COMPASS** attempts to locate similar code segments *across several code bases* with the goal of improving *performance*, while code clone tools generally aim to locate similar code segments *within a code base* to improve *maintainability*. The increased code size and the need to look across code bases places different stresses on the signature representation and precludes the possibility of adopting the internal representations used for detecting code clones.

§ **Related Work Analysis** Textual [2], Abstract Syntax Tree (AST) [3, 12] and token-based [15] representation methods for detecting code clones have been proposed. These encodings are effective for catching "cut and paste" statements in a code base, but fail when the code is semantically the same but syntactically different. For example, for and while loops with slightly different syntactic bodies may elude clone detection. Clearly, these representations do not have sufficient universality to be used as signatures in a community knowledge sharing system like **COMPASS**. The Program Dependence Graph (PDG) [9], which can be described as a graph of program statements (nodes) and dependencies between the statements (edges), has been proposed to overcome the syntactic limitations of ASTs [10, 13]. In this representation, semantically identical code regions result in isomorphic portions, so detecting code clones reduces to the problem of colored subgraph isomorphism. This process is intractable in the gen-

eral case, but solvable for small graphs (typically procedure level PDGs) in the absence of aliases. In the presence of aliases, however, PDGs may suffer from low specificity. For instance, when an ambiguous node is present in a graph, all nodes following the ambiguous node in the graph should include an edge to the ambiguous node indicating a possible dependence. As the number of ambiguous nodes increases, the number of ambiguous edges increases, consequently diminishing the power of the representation to uniquely identify code segments across large and diverse code bases.

§ **Efficient COMPASS Signatures** To overcome the specificity shortcomings of PDGs, we introduce a new program abstraction called the *Segment Dependence Graph* (SDG). Like a PDG, an SDG represents dependencies between statements; but unlike the PDG, the SDG has no ambiguous edges (and thus improves specificity). In a SDG, we sidestep the problem of ambiguity of static alias analysis techniques by annotating the edges with a purely dynamic measurement of dependencies. Let us consider an example to better explain SDGs and then illustrate the benefits over PDGs. Consider a simple program segment with N SSA-style statements [6] $S_1, \ldots, S_N$ where S_i denotes that statement S_i executed after S_{i-1}. Also assume that the statement S_3 depends 2 times on S_2 and 3 times on S_1 during some execution of the segment. The SDG for this segment will be constructed such that the edges to S_3 from S_1 and S_2 will be annotated with the fraction of the times these dependences occur, i.e., 0.4 and 0.6, respectively. To illustrate the differences from a PDG, consider the case where the statement S_2 is ambiguous, i.e., the compiler cannot statically determine if the statements following S_2 depend on S_2; to denote this the PDG for this segment will have an edge from S_2 to all statements after S_2 but the SDG will not. While in theory these run-time annotations hold only for a given input, in practice the dependency information tends to be fairly stable across inputs. We are not the first to make this observation: compiler researchers have proposed using run-time dependency information for profile-driven speculative optimizations [22] and run-time system builders have implemented some of these optimizations in production systems [7]. COMPASS, which does not have the same strong correctness criteria as compilers, will also stand to benefit from this kind of representation.

3.2 Search and Retrieval

COMPASS's knowledge database stores SDGs as $< key, value >$ pairs, in which the *key* corresponds to the unoptimized before version and the *value* is the optimized after version. Since SDGs are graphs, queries employing only a traditional relational database schema, which are typically optimized for flat text and numerical data, are unlikely to be very efficient. Further, it may be desirable to find an approximately matching SDG if an exactly matching SDG is not available in the data store, because minor variations in SDGs for the same program segment are to be expected. Minor variations can be a result of different environmental factors such as differences in caller-callee save conventions, register and memory allocation procedures, etc. Additionally, when more than one prospective advice (value) exists for a given SDG (key), the data store must rank the solutions and return what the system considers to be the most useful solution(s) in order.

§ **Related Work Analysis** We do not know of any published work that has encountered or attempted to solve this problem. The closest in spirit to COMPASS's proposed code search system is the string-based regular expression search service provided by the Google Labs Code Search Engine [4]. Based on the authors' trial of that system (its architecture has not been published), the service appears to be little more than a sophisticated implementation of the unix "grep" utility over code repositories on the web, with no perceivable ranking of search results.

§ **Overview of Database Operations** Since graph isomorphism checks are computationally expensive, to enable quickly searching through potentially hundreds of thousands of SDGs, COMPASS uses a series of pre-filters to first narrow the number of items that have to be searched. The pre-filters can be grouped into two categories: (a) Environment filters and (b) Graph intrinsics filters. Environment filters use some features of the target machine specification (such as the instruction set architecture and cache configuration), the target compiler, the libraries available on the target machine, and the source language to reduce the search scope. The features required for environmental filtering are included by the matcher query tool as part of the segment signature. The graph intrinsic filters are based on the number of nodes and edges in the query SDGs.

To pick one or a few candidate SDGs from the pre-

filtered SDGs, COMPASS uses a new ranking method we call "coderank". Our coderank metric is based on the intuition that isomorphism alone is insufficient to recommend a SDG; some times an imperfectly matching SDG but with higher perceived usefulness may be preferred, which may be the case if the perfect matching SDG is untrustworthy and perhaps even intentionally seeded into the database (an obvious problem with any community resource). The coderank is computed based on three distinct characteristics: (a) the structural similarity between the key and query SDG, (b) the dependence annotation similarity between the key and query SDG, and (c) the perceived importance of the value SDG corresponding to a key SDG (similar to the pagerank algorithm [19]). The structural similarity is computed as an Isomorphism score (I-score) and is defined as the fraction of subgraphs that are similar between the two graphs. The annotation similarity is computed as the Euclidean distance (E-score) between the dependence annotation on similar edges in the two graphs. The perceived importance score (P-score) is based on a combination of factors including the improvement in performance on the target machine (transmitted when the learner accepts reprofiling results) and the number of users using a particular advice (both gurus providing and learners accepting). An open question is to determine the weightings of the P, E and I-scores for determination of the most effective coderank.

3.3 Program Presentation Innovations

One of the core goals of COMPASS is to provide advice in a format easy for the learner to use. COMPASS's advice is presented in the form of a code "sketch" – an outline that shows the main control flow and data structure changes required for parallelism – closely matching the source code that the user would like to parallelize. The key to such a sketch is converting the parallelized SDGs into optimized code sequences tailored to the user's code context. The challenge is that an SDG can be translated to source code in many ways, as it does not preserve syntactic information. For example, it is easy to detect a loop from a SDG but difficult to say whether the loop is a `for` loop or a `while` loop. Similarly, it may be possible to say two arrays are being added but may not be possible to guess the names of the arrays when there are more than two choices in the context. For an effective usable sketch, a method for inferring ambiguous syntactic information is required.

§ **Related Work Analysis** Decompilers [8], Source-to-Source translators [1] and Pretty print tools [18] are commonly used to translate between compiler formats. The difficulty of translation, and the subsequent quality of the output, depends on how much high-level syntactic information is preserved in the input format. Pretty printing tools and source-to-source translators start with input that has lot of syntatic information, and generally produce reasonable output; decompilers start with assembly format in which there is very less syntactic information and produce output suitable for mature, patient users. The conversion tool used in COMPASS is different from the above since COMPASS's goal is to create only a "sketch" corresponding to the SDG, and not the full source code. Further, the sketch is created from *two* input sources: an existing source file (with full syntactic information) and a parallelized SDG (with little syntactic information), whereas previous works have only one input format with either full syntactic information or none.

Our work is complementary to the work on sketching languages [23, 24], which are meta-languages that permit users to write incomplete programs that can be automatically completed using sophisticated static analyis. Sketch analysis tools expect the user to specify program invariants and then use that invariance information along with the program structure to create fully specified programs. COMPASS and sketching languages can potentially enhance each others' utility. Other sketching schemes require humans to prepare the sketches. COMPASS, in contrast, has the ability to automatically generate sketches. Similarly, once COMPASS has a sketch, if the learner's program is annotated with invariants, a sketch compiler may be able to turn the sketch into code and thus enhance productivity further.

§ **Sketch Generation** COMPASS uses a multi-step procedure to create a sketch from an SDG. Our algorithm operates on: (1) the unoptimized source code and its SDG (called the program SDG) and (2) the response from the data store, which contains (a) the recommended SDG (the *value* from some $< key, value >$ pair), called the advice SDG, and (b) the SDG corresponding to the advice SDG in the data store (the key from that pair), called the key SDG (which is not necessarily the same as the program SDG sent by the matcher tool). First, the program SDG is annotated with the missing syntactic information based on source code

analysis. Then the annotated program SDG is compared to the key SDG and semantically similar statements between the two SDGs are used to deduce a mapping for variable names in the key SDG. These variable names are then used to annotate the advice SDG by simple substitutions. When the advice SDG has been customized to the maximum possible extent based on the information available in the unoptimized source code, the next step is to convert the advice SDG into higher level source code. The control flow structures are determined based on the structure of the graph followed by statement substitution in the control flow blocks. Finally, the sketch (with some possibly undeterminable portions) is displayed to the user as a graphical overlay.

4 Ongoing Efforts

The pilot implementation of **COMPASS** is capable of analyzing only C/C++ code and is built around several open source tools available on the Linux platform. We use the gcov [16] coverage tool to identify frequently executed regions; an Eclipse [11] plugin for adjusting the hotspot regions identified by gcov; LLVM [14], a compiler research infrastructure, to generate signatures; and an SQL database for storing and matching the signatures. With the current very preliminary infrastructure, **COMPASS** can already analyze enterprise-scale projects such as the open source javascript engine v8 [5]. **COMPASS** identifies the most profitable regions for parallelization, displays these regions graphically to the user, creates signatures from the code, sends signatures to a database, and obtains matches when one or more exists. We seeded the database with a few hand-coded sample optimizations. Efforts are underway to build the sketching capabilities and a more sophisticated database schema. We hope to open the system for trial release to a select group of users in the last quarter of 2009. Prospective users are requested to register for an account at `http://compass.cs.columbia.edu`.

5 Conclusions

The trend towards CMPs (and implicitly the emphasis on concurrency for performance improvements) is likely to continue for the foreseeable future because of VLSI technology trends. The move to multicores has forced software engineers to either reengineer their code for parallelization or accept significant performance slowdown (because multicores have lower clock frequency than their uniprocessor predecessors). We have described our vision and provided implementation details for our pilot prototype of an Internet-scale community-based system for rapid propagation of "best practices" for parallelization of sequential code. We envision that **COMPASS** can dramatically increase programmer productivity when attempting to leverage today's and tomorrow's computer hardware, and thus retain performance improvements in line with historical trends.

COMPASS uses an alias-free program segment representation as its basis for the code search engine used to locate parallelization solutions contributed by users. The results from the search engine are ranked using heuristics and the most relevant parallelization strategy is returned to the user in a custom format as a starting point for parallelization. We believe that **COMPASS** as currently designed is particularly suitable for rapid incremental parallelization of sequential code for CMPs. Alternative approaches such as complete re-writes of applications using new languages, or automatic compiler analyses, may prove preferable in the long run but seem less likely to achieve practical adoption and success within the immediate time horizon.

A caveat for any such community system is the need to address privacy and intellectual property concerns. This is one reason why no syntactic information is retained in SDGs. Users can participate anonymously, and retrieve from **COMPASS** without ever contributing to it. A commercial enterprise could potentially set up their own private instance of **COMPASS** separate from the volunteer community resource hosted at Columbia (we are currently seeking a sponsor for longer term maintenance).

COMPASS is not just a reactive solution to fundamental changes in computer architecture; it provides an infrastructure that can proactively influence and aid in the design of new computer systems. For example, in the steady-state **COMPASS** could be data-mined to determine the most frequently requested optimizations and that information utilized by computer architects to create new hardware extension units or by compiler writers to generate targeted compiler analyses that speedup these frequently requested (executed) regions.

6 Acknowledgements

We would like to thank Nipun Arora and Bing Wu for work on the **COMPASS** pilot project and Ravindra Babu Ganap-

athi for help with the hand-optimized code samples. The Programming Systems Laboratory is funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1. The Computer Architecture Laboratory is funded in part by AFRL FA8650-08-C-7851.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 86, Washington, DC, USA, 1995. IEEE Computer Society.

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[4] Google Corporation. The google code search engine. `http://http://www.google.com/codesearch`.

[5] Google Corporation. V8 javascript engine. `http://code.google.com/p/v8/`.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[7] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Mike Van Emmerik and Trent Waddington. Using a decompiler for real-world source recovery. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 27–36, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[10] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 321–330, New York, NY, USA, 2008. ACM.

[11] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The eclipse 3.0 platform: adopting osgi technology. *IBM Syst. J.*, 44(2):289–299, 2005.

[12] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[15] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[16] GNU GPL License. Gcov: Gnu coverage tool. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[17] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[18] Dereck C. Oppen. Prettyprinting. *ACM Trans. Program. Lang. Syst.*, 2(4):465–483, 1980.

[19] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[20] Daniel Reed. The law of the pack. *Harvard Business Review*, pages 2–3, 2001.

[21] Simha Sethumadhavan and Gail E. Kaiser. Rapid parallelization by collaboration. Technical Report CUCS-002-09, Department of Computer Science, Columbia University, New York, NY, January 2009.

[22] Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. *SIGPLAN Not.*, 41(11):416–425, 2006.

[23] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 167–178, New York, NY, USA, 2007. ACM.

[24] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGPLAN Notices*, 41(11):404–415, 2006.

[25] E. Stewart. *Intel Integrated Performance Primitives: How to Optimize Software Applications Using Intel IPP*. Intel Press, 2004.

[26] James Surowiecki. *The Wisdom of Crowds*. Anchor, 2005.