AFRL-RY-WP-TR-2011-1213

ADAPTIVE ENVIRONMENT FOR SUPERCOMPILING WITH OPTIMIZED PARALLELISM (AESOP)

Jonathan Rosenberg

BAE Systems, Inc.

SEPTEMBER 2011
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

*//Signature//

KERRY HILL, Program Manager
Advanced Sensor Components Branch
Aerospace Components Division

//Signature//

BRADLEY J. PAUL, Chief
Advanced Sensor Components Branch
Aerospace Components Division

//Signature//

BRADLEY CHRISTIAN, Lt Col, USAF
Deputy Division Chief
Aerospace Components Division
Sensors Directorate

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YY)* September 2011	**2. REPORT TYPE** Final	**3. DATES COVERED *(From - To)*** 09 March 2009 – 31 July 2011

4. TITLE AND SUBTITLE ADAPTIVE ENVIRONMENT FOR SUPERCOMPILING WITH OPTIMIZED PARALLELISM (AESOP)	**5a. CONTRACT NUMBER** FA8650-09-C-7918
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62303E
6. AUTHOR(S) Jonathan Rosenberg	**5d. PROJECT NUMBER** ARPR
	5e. TASK NUMBER YD
	5f. WORK UNIT NUMBER ARPRYD1D
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) BAE Systems, Inc. 6 NE Executive Park Burlington, MA 01803	**8. PERFORMING ORGANIZATION REPORT NUMBER** TR-2717
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, Sensors Directorate, Wright-Patterson Air Force Base, OH 45433-7320, Air Force Materiel Command, United States Air Force / Defense Advanced Research Projects Agency/ Information Processing Techniques Office (DARPA/IPTO) 3701 N. Fairfax Drive Arlington, VA 22203-1714	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/RYDI
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-RY-WP-TR-2011-1213

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
DARPA PAO case number DISTAR 18513; Clearance Date: 20 Nov 2011. This report contains color.

14. ABSTRACT
Moore's Law would have ceased applying if all new chips had not become multicore a decade ago. Demand for higher performance continues unabated but to achieve that performance, applications must employ parallelization. Programmers are not equipped for that and most codes have no notion of parallelization in them. Meanwhile, compilers are increasingly complex and highly tuned to their target platform, which evolves too rapidly for them to keep up. Hence, research is needed to address these problems. AESOP addresses all these issues: it adapts to any hardware it is to target, automatically parallelizes serial codes as this project ended. We recommend DARPA create new programs focused on the problems of parallelization and tools to support it as a matter of high priority so that military systems can meet the increased demands placed on them.

15. SUBJECT TERMS
compiler, parallelizing, optimizing, multicore, software pipelining, affine, LLVM, system characterization, self-tuning, iterative

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	SAR	90	Kerry Hill
Unclassified	Unclassified	Unclassified			**19b. TELEPHONE NUMBER** *(Include Area Code)* N/A

Table of Contents

Section **Page**

List of Figures iii
List of Tables iii
Acknowledgements iv
1.0 Summary 1
  1.1 Problem 1
  1.2 Results 2
  1.3 Conclusions 2
  1.4 Recommendations 3
2.0 Introduction 4
  2.1 System characterization loop 5
  2.2 Continuous optimization loop 5
  2.3 Learning and reasoning 6
3.0 Methods, Assumptions, and Procedures 7
  3.1 Leveraged industry standard open-source frameworks 7
    3.1.1 LLVM is the foundation for AESOP 7
    3.1.2 Other (non-open-source) research tools utilized 12
  3.2 Parallel research thrusts driven by PIs at each sub-contractor. 13
    3.2.1 Characterization (Alan Sussman – University of Maryland) 13
    3.2.2 Affine parallelizer (Rajeev Barua – University of Maryland) 13
    3.2.3 Non-affine parallelizer (David August – Princeton University) 15
  3.3 Divide-and-conquer then re-unite and integrate. 20
  3.4 A “spike” approach to integration was used 20
  3.5 Leveraged work of other AACErs to the maximum extent 21
4.0 Results and Discussion 22
  4.1 System Characterizer 22
    4.1.1 Instruction performance 22
    4.1.2 Cache size and characteristics 24
    4.1.3 Contexts 25
    4.1.4 Pairwise communication costs 26
    4.1.5 Barrier modeling 26
    4.1.6 Dynamic System Characterization 27
  4.2 Critical data structures 28
    4.2.1 The Control Flow Graph (CFG) 29
    4.2.2 The Program Dependence Graph (PDG) 29
  4.3 Negotiator 32
    4.3.1 Introduction 32
    4.3.2 Practical results 35
    4.3.3 Negotiator Example 36
  4.4 LLVM IR Processing and the Negotiator 39
  4.5 Affine Parallelizer 42
    4.5.1 Use of static system characteristics 44
    4.5.2 Affine transformations implemented 45

4.5.3 Results 48
4.6 Software Pipelining Parallelizer 50
4.6.1 Decoupled Software Pipelining (DSWP) 51
4.6.2 Parallel Stage-Decoupled Software Pipelining (PS-DSWP) 53
4.6.3 Enhancing the Applicability of Pipeline Parallelism 55
4.6.4 Commutative Annotations 55
4.6.5 Privatization 56
4.6.6 Reductions 57
4.6.7 Multi-Threaded Code Generator (MTCG) 57
4.6.8 Queue Implementation 66
4.6.9 Exploiting System Characterization Parameters 68
5.0 Conclusions 70
6.0 Recommendations 71
BIBLIOGRAPHY 72
LIST OF ACRONYMS, ABBREVIATIONS, AND SYMBOLS 75
INDEX 80

List of Figures

Figure **Page**

1. The Overall Design of the AESOP Compiler 5
2. Block Diagram of LLVM 8
3. LLVM Integration Points for AESOP 10
4. LLVM and the AESOP Compiler 12
5. Affine Parallelization Flow Diagram 14
6. DOALL Execution of Simple Loop 15
7. DOACROSS Parallelization 17
8. DSWP Breaks Iterations Across Threads 18
9. PS-DSWP Replicates Large Pipeline Stages 19
10. Listing 1: A Sample Nano-Benchmark 23
11. Listing 1: A Sample Nano-Benchmark (continued) 24
12. The CFG of a Function 29
13. HCI in Negotiator 33
14. Listing 2: Using the Negotiation Framework to Extract Parallelism: Original Loop 37
15. Listing 3: Loop Transformed with DSWP 38
16. Listing 4: Second Stage After Loop Skewing 39
17. AESOP's IR Processing 41
18. Decision Algorithm for Affine Parallelizer 42
19. Software Pipelining Parallelizer 50
20. The DSWP Algorithm 51
21. Property 1: Valid DSWP Assignment 52
22. Property 2: Valid PS-DSWP Assignment 53
23. PS-DSWP Thread Partitioning Algorithm 54
24. Listing 5: Simplified Version of the Main Loop Body of HMMER with Commutative Annotation 56
25. Pseudocode for the MTCG Algorithm 58
26. Example of the MTCG Algorithm 60
27. Definition 1: Relevant Basic Block 61
28. Property 3: Basic Communication Placement 62
29. Liberty Queues 67

List of Tables

Table **Page**

1. Speedups With and Without Transformations for Polybench Benchmarks 49
2. Example Using Affine and Non-Affine Transformations 49

Acknowledgements

The following institutions joined BAE Systems as part of our AESOP team and we would like to acknowledge their contributions to the work described herein:

Institution	*Principal*
University of Maryland	Rajeev Barua & Alan Sussman
Parakinetics, Inc	David August
Princeton University	David August
University of Michigan	Scott Mahlke
SRC Computers, LLC	Ken Nicholson
Cray, Inc.	Dan Poznanovic

1.0 Summary

1.1 Problem

When Intel first came out with the dual-core chip, it was the beginning of the end of the single Central Processing Unit (CPU) on a chip era. Because they could no longer keep speeding up clock speeds of a single CPU, the only way to improve performance and stay on Moore's law[1] was to simply put a second (and soon 4, 8 16, etc.) CPUs or cores on a single silicon die. But while doing this they also had to slightly slow down the clock speed of each CPU to avoid dramatically higher power requirements for the overall chips. This meant that applications that used only one core were actually seeing a degradation of their performance.

By far the vast majority of the world's application codes are not written to take advantage of any type of parallelism in the hardware on which they run. Parallel programming, even from scratch in new applications, is very difficult and the vast majority of programmers are not proficient at it. Those that are proficient at parallel programming would have trouble beating effective automation at the same task. Humans have to completely rethink and rearchitect an application to get significantly better results than the best automatic parallelizing compilers today.

Compilers have become extremely complex systems; so complex that they cannot keep up with the rapid evolution going on in the hardware domain. By the time a compiler is hand-tuned to a new generation of hardware, the next generation of that hardware is released and the cycle of compiler tuning has to begin anew.

Virtually every domain – and especially within all military domains that are the purview of the Defense Advanced Research Projects Agency (DARPA) – requires more and more complex applications that could benefit from the much higher performance parallel hardware currently available off the shelf. Applications that include any sort of signal processing, video processing, or real-time data analysis, to mention just a few algorithm types, require (and are highly suitable for) high performance parallel hardware.

The military always has and will continue to push the limits of each generation of hardware. All systems being developed for the military depend on software and many systems now have software representing more than 50% of their cost. This, combined with the gaps between existing serial codes and the latest hardware and custom-tuned compilers mentioned already, is creating a Department of Defense (DoD) crisis. In many areas, this applies equally strongly to the commercial sector as well. This is a DARPA-hard problem worthy of strong efforts to solve it as Architecture Aware Compiler Environment (AACE) was designed to do.

[1] **Moore's law** states that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years. This trend has continued for more than half a century and is expected to continue at least throughout the current decade.

Ultimately, all of this has led to a problem statement that seeks new compiler technology that:

a) Can adapt to parallel hardware as it evolves and improves;

b) Can efficiently, accurately, and effectively find and extract parallelism from serial codes; and

c) Can learn and adapt to different hardware, different data sets and changing environments even while an application is executing.

1.2 Results

The extended AESOP team took on this challenge.

First, we developed a system characterization system that both statically, at compiler installation time and dynamically at application run time, probes and learns the detailed parameters of the hardware and environment and uses this information to tune the AESOP compiler to any new platform the compiler is required to target.

Second, our compiler is built on top of the industry standard open-source framework called Low Level Virtual Machine (LLVM) (llvm.org) that is also the basis for compilers used by Apple and Adobe. This led to a working compiler years sooner than possible if it were built from scratch. Within this framework, we developed a very clean architecture that supports multiple parallelizers (including future ones we have not yet even thought of) that use the simple but powerful plugin architecture of LLVM.

Third, our negotiator is a novel mechanism for choosing which parallelizer to apply to each block of code. This negotiator is also our vehicle for integration of all parallelizers into a clean unified compiler. It also represents significant innovation in the area of human-machine interaction in the way it includes the human in attempting to break dependencies in the code that no automated system can but which make 2X, 5X, even 10X differences in performance.

Fourth and finally, two parallelizers were built—an affine parallelizer and a software pipelining parallelizer—and plugged into the negotiator framework. The affine parallelizer is very effective at automatically parallelizing linear loops even when quite complex. The software pipelining parallelizer is as advanced as any in the world at finding and exploiting parallelization in serial codes.

We had just begun to use a “spike” approach to show that we had a single integrated compiler that could compile real, albeit trivial, applications when the project was cancelled.

1.3 Conclusions

Our assessment is that the problem statement as captured here and in the original DARPA Broad Area Announcement (BAA) is correct. If anything, the time that has passed since the original BAA has proven the problem is worse than originally characterized. With the push for Graphics Processing Units (GPUs) to become General Purpose-GPUs (GP-GPUs) and the continued use of Field-Programmable Gate Arrays (FPGAs)—which are exceedingly difficult to program—the need for high performance parallel hardware is making design teams go to extreme lengths. And yet, the ease of writing applications that make effective use of all the parallel resources available to them has not improved. The crisis DARPA—and many in the commercial sector—has called out is real, is pressing and is not going away. The approach taken by the AESOP team to attack this problem through an automatically adapting and parallelizing compiler appears very effective based on early results.

1.4 Recommendations

DARPA should quickly resurrect the effort in a different form, with a full parallel programming program a major thrust. A full complement of research into compilers, languages, debuggers, complete Interactive Development Environments (IDEs) and even training should be created to help address this identified and critical computing crisis. Until then this crisis will continue to affect military systems detrimentally until it is thoroughly addressed.

2.0 Introduction

The initial goal for the AACE program as stated by DARPA in its BAA:

To reduce the burden on programmers and to make more effective use of the underlying hardware, a completely new approach to compilers is required to resolve these formidable problems. Application software is rapidly becoming one of the DOD's costliest and error-prone areas. The envisioned Architecture-Aware Compiler Environment (AACE) Program has the potential to dramatically reduce application development costs and labor; ensure that executable code is optimal, correct, and timely; provide the full capabilities of computing system advances to our warfighters; and provide superior design and performance capabilities across a broad range of applications.

To meet this goal the AESOP team developed novel compiler technology that brings practical automatic parallelization into mainstream software development, reflecting the fact that people program using conventional serial programming languages (e.g. C, C++, Fortran) rather than specialized languages invented anew for each parallel machine built. Targeted architectures include the mainstream parallel architectures built recently and expected to come out over the next few generations of computer architecture. Our prototype compiler supports only shared-memory Multiple Instruction Multiple Data (MIMD) systems, but that limitation is not fundamental to our approach. Although the vast majority of existing codes are serial and will remain so for quite some time, some codes have been carefully crafted to support a style of parallel programming based on various message-passing technologies. Thus, AESOP supports codes employing OpenMP or Message Passing Interface (MPI) annotations and will not destroy the parallelism called out by those annotations.

Three key design thrusts drove our development:

- System Characterization automatically extracts key descriptive elements about the computing platform to inform the compiler's optimizations (e.g. memory architecture, core configuration).

- Automatic Parallelization automatically parallelizes sequential codes, extracts fine-grained parallelization from explicitly parallelized code, and allows the programmer to add lightweight semantic annotations enabling even more significant levels of parallelization.

- Continuous Optimization and Learning exploits parallel hardware to conduct multiple experiments to learn how to estimate performance from system characteristics. These key elements are divided into three main areas shown in Figure 1. We describe them in more detail now based on that diagram.

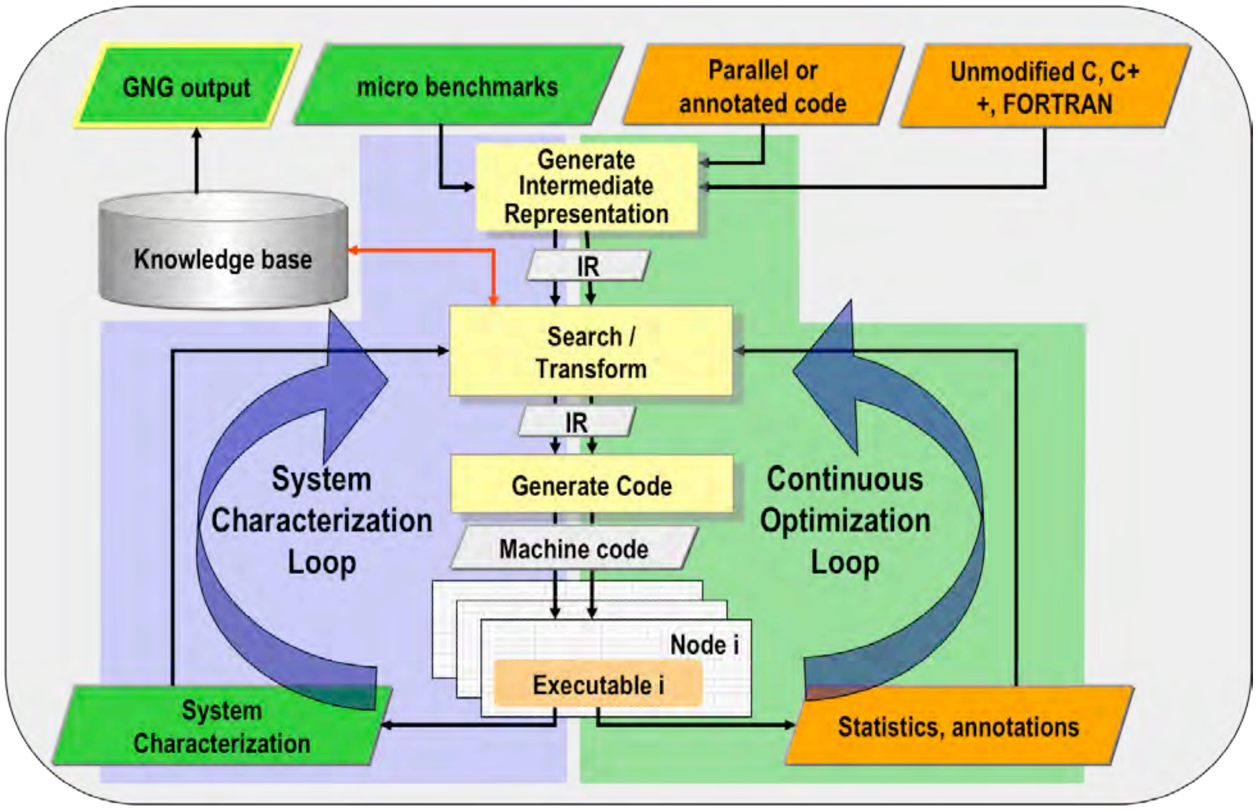


Figure 1. The Overall Design of the AESOP Compiler

AESOP uses the System Characterization Loop results to increase the efficiency of its Continuous Optimization Loop for auto-parallelizing, compiling and running applications on heterogeneous parallel hardware.

2.1 System characterization loop

The System Characterization Loop is highlighted on the left in Figure 1. System characterization is seeded by micro benchmarks (green box input to Generate Intermediate Representation) and iterates, driven by a nonlinear optimization search engine. Derived system characteristics (bottom left box) drive and increase the efficiency of the Continuous Optimization Loop for auto-parallelizing, compiling and running parallel applications.

2.2 Continuous optimization loop

The Continuous Optimization Loop (diagram right) encompasses our dynamic runtime system, which includes applying transformations (e.g. loop unrolling, code hoisting, application of concurrent threads) and instrumentation to code, monitoring the results of the transformations and instrumentation, and then applying further transformations. Knowledge-based search chooses the next set of optimizations to explore the massive search space of program transformations.

2.3 Learning and reasoning

The Learning and Reasoning element consists of the persistent Knowledge Base (upper left), its inputs—which come from both system characterization and application instrumentation—and its query interface Application Programming Interface (API) (red line to Search / Transform). The Knowledge Base tracks relations between code fragment features (e.g. operation count), optimization parameters (e.g. loop unrolls) and system characteristics (e.g. L1 cache size) and continuously refines learned classifications. The learned correlations among machine, code, transform characteristics and performance results are inputs to the Search/Transform module to inform the next positions in the search space of program transformations.

These technologies combine to allow AESOP to automate the optimization and parallelization of both scientific and general purpose applications, to increase programmer productivity through automation, to quickly produce detailed system characterizations in order to seed the optimization search, and to adapt automatically to novel system configurations and future parallel hardware architectures.

Key innovations we are contributing to compiler design and to the growing body of knowledge around parallelization include the following list of items:

- Linear programming approach to system characterization minimizes the number of micro-benchmark runs needed to reach >90% accuracy.

- Automatic parallelization of general-purpose sequential code enables >10X productivity for full software life cycle and range of system configurations.

- Run-time system uses efficient heuristic search to adapt parallelization and optimization to the system to achieve >20% performance increase.

- Learning from results of optimization seeds iterative compilation and transfers results across runs, programs and even hardware.

- Design combines all these elements seamlessly including adapting codes to new hardware systems with zero code modifications.

3.0 Methods, Assumptions, and Procedures

3.1 Leveraged industry standard open-source frameworks

Wherever practical, AESOP utilizes open-source frameworks to build out the compiler. This minimized wasted effort on superstructure and allowed the team to focus on just our value-add and ultimately to build a working compiler much faster than possible otherwise.

AESOP's architecture is designed to promote broad applicability and user acceptance. Because it is based on the Low Level Virtual Machine (LLVM) compiler framework, and adopts its unencumbered open source licensing, combined with LLVM's plug-in architecture, AESOP is an ideal platform for researchers and commercial entities. LLVM has an active community of open source contributors as well as commercial users (such as Apple and Adobe). The AESOP project will contribute to the LLVM project, but will also benefit greatly from the significant efforts of the many current and past contributors. Leveraging LLVM substantially reduces the risk of the AESOP project and allows us to target a wide range of source languages and target hardware. Like LLVM, AESOP employs an open source, plugin-based architecture creating a marketplace for open source contributions, and commercial software and hardware vendors. Our architecture for AESOP, and our use of the same unencumbered open source licensing as LLVM, is carefully crafted to invite novel optimizations developed outside of our own research groups. Furthermore, some team members themselves intend to leverage the commercial potential of AESOP into their commercial offerings.

3.1.1 LLVM is the foundation for AESOP

LLVM is an open-source compiler development begun by Chris Lattner at University of Illinois at Urbana-Champaign (UIUC). Documentation and sources for LLVM can be found at http://LLVM.org. A simple diagram of LLVM is shown in Figure 2.

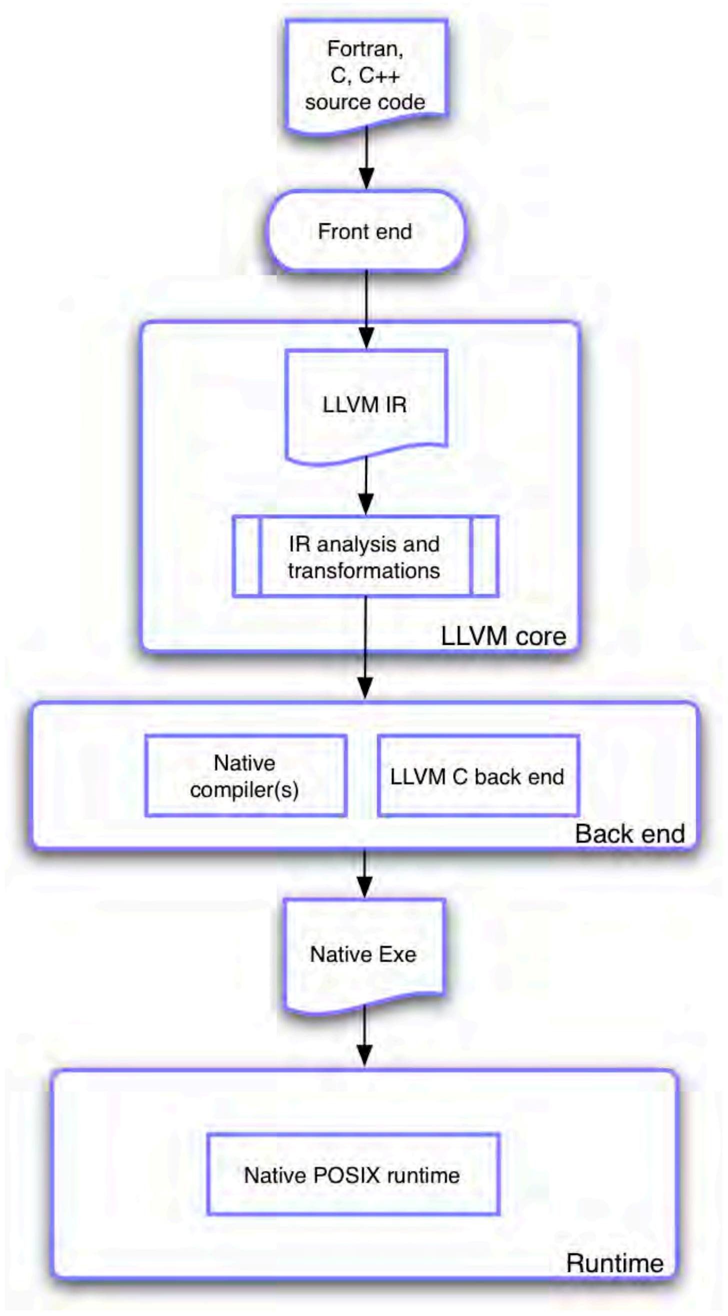


Figure 2. Block Diagram of LLVM

Standard LLVM takes C, C++ or Fortran code as source code input, processes and transforms it and produces either native binary code for the processors it has a back-end for, or ANSI C that can then be processed by a native C compiler for the target machine.

3.1.1.1 What is LLVM?

The LLVM Project is a collection of modular and reusable compiler and tool chain technologies. The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs. These libraries are built around a well-specified code representation known as the LLVM intermediate representation (LLVM IR). The LLVM Core libraries are well documented, and it is particularly easy to build or port an existing compiler using LLVM as an optimizer and code generator.

At a high level, LLVM can be understood as a modern rewrite of the GNU Compiler Collection (GCC). It specifically attempts to duplicate the capabilities and user interface of GCC 4.2, supporting a broad set of language front-ends and an even larger array of target instruction set architectures.

Structurally, LLVM is a framework that efficiently supports modular analysis and transformation passes over a common Intermediate Representation (IR) of a program. The IR is low-level, in that it represents the program in what might be seen as a simple 3-register RISC instruction set augmented with an infinite register pool to permit Static Single Assignment (SSA), and it also retains sufficient type information to enable accurate alias analysis. The IR can be retained through linking and execution of the program, allowing optional inter-procedural analysis (including libraries, even across languages), just-in-time code generation, and runtime profile-driven optimization if desired.

LLVM appears to have replaced GCC as the default basis of new compiler development within the open-source community; Lattner is now continuing development at Apple; Advanced Micro Devices (AMD) and Adobe have also adopted it, and the list of active contributors to the source pool is impressive. Recently, LLVM achieved the 100,000 revisions milestone attesting to its active and large community. The AESOP team will integrate into the community, and contribute to LLVM core.

LLVM is designed to be extended and tuned, with self-registering analysis and transform passes implemented as dynamically linked libraries, and a control framework that attempts to optimally schedule the execution of passes over selected code regions to minimize re-computation of analyses as the code is transformed.

The way in which AESOP was inserted (blue boxes) into the LLVM flow is shown in Figure 3.

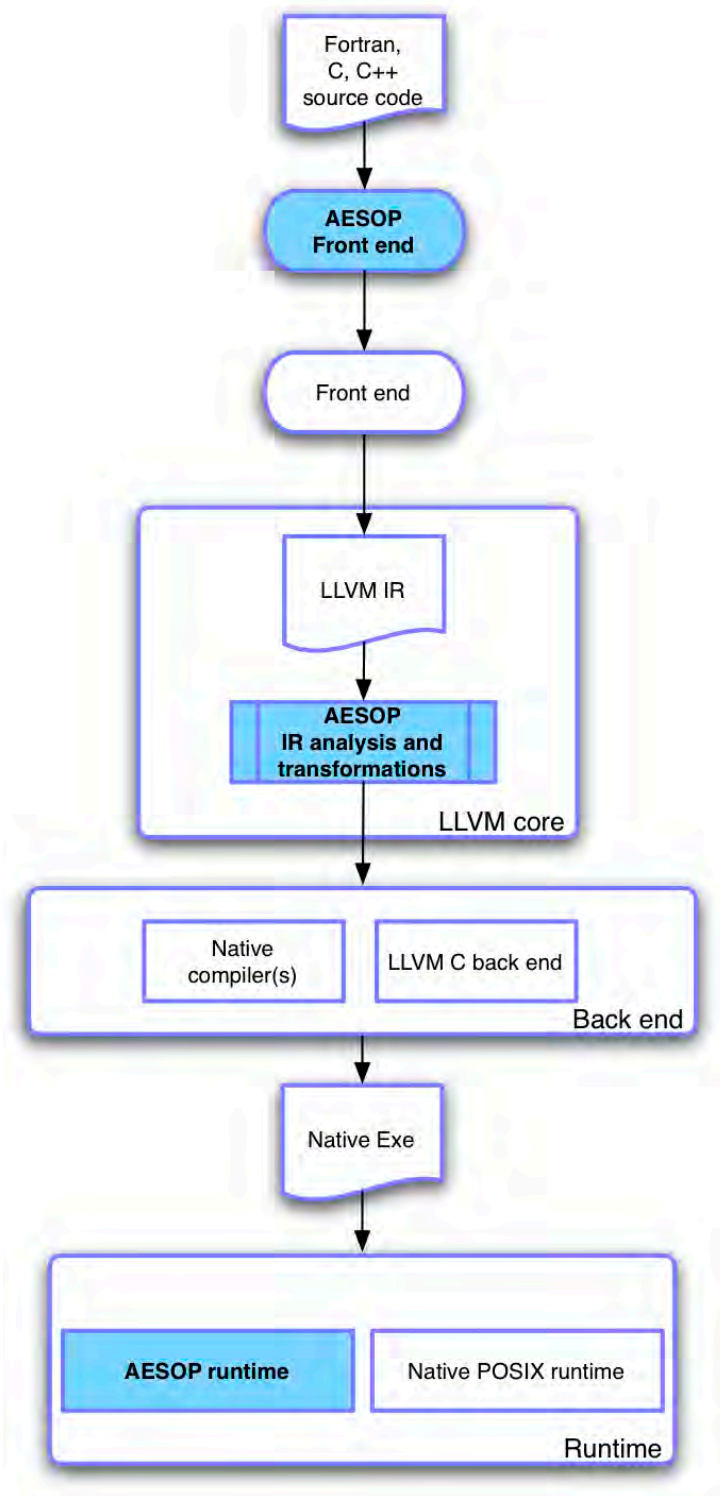


Figure 3. LLVM Integration Points for AESOP

LLVM with AESOP points of integration shown. Places where AESOP has integrated into LLVM are shown in blue. Additional components external to LLVM proper will be shown in later architecture diagrams.

3.1.1.2 Why Does AESOP Use LLVM?

The LLVM infrastructure is designed to be modular. Each of the optimization passes is a self-contained transform that takes LLVM IR as input and produces it as output. Any combination of the optimizations can be run in any order; sometimes it's appropriate to run one more than once. LLVM aims to allow optimizations to be run at any time. When a module is compiled to the IR, the first set of optimizations runs. Then, when it's linked with other modules, it can be optimized again. This functionality is used by the OpenGL Shading Language (GLSL) implementation on newer versions of Mac OS X.

3.1.1.3 How Does AESOP Use LLVM?

LLVM has a plugin architecture. LLVM utilities, like **`opt`**, load the shared objects specified by **`-load`** on the command-line. Passes from a shared object are requested on the command-line in exactly the same way as LLVM's built-in passes. All of LLVM's passes implement the same plugin interface. The only difference is that some are statically linked and others dynamically linked.

Passes are scheduled on the command-line. For example:

```
opt foo.bc -loopsimplify –pdg
```

builds a Program Dependence Graph (PDG) after running loop simplification. However, both **`loopsimplify`** and **`pdg`** have dependencies that LLVM will automatically satisfy. The real order of passes will be:

```
-domtree -loops -loopsimplify -postdomtree -pdg -preverify -verify
```

`domtree` and **`loops`** are needed for **`loopsimplify`** and **`postdomtree`** is needed for **`pdg`**. The passes **`preverify`** and **`verify`** run after all other passes to ensure that the IR is valid before serialization.

The interface required for adding a new analysis or transform pass to LLVM is documented at http://LLVM.org/docs/WritingAnLLVMPass.html.

LLVM with the major components of AESOP, including the knowledge base and a block-diagram representation of AESOP's IR processing, is shown in Figure 4.

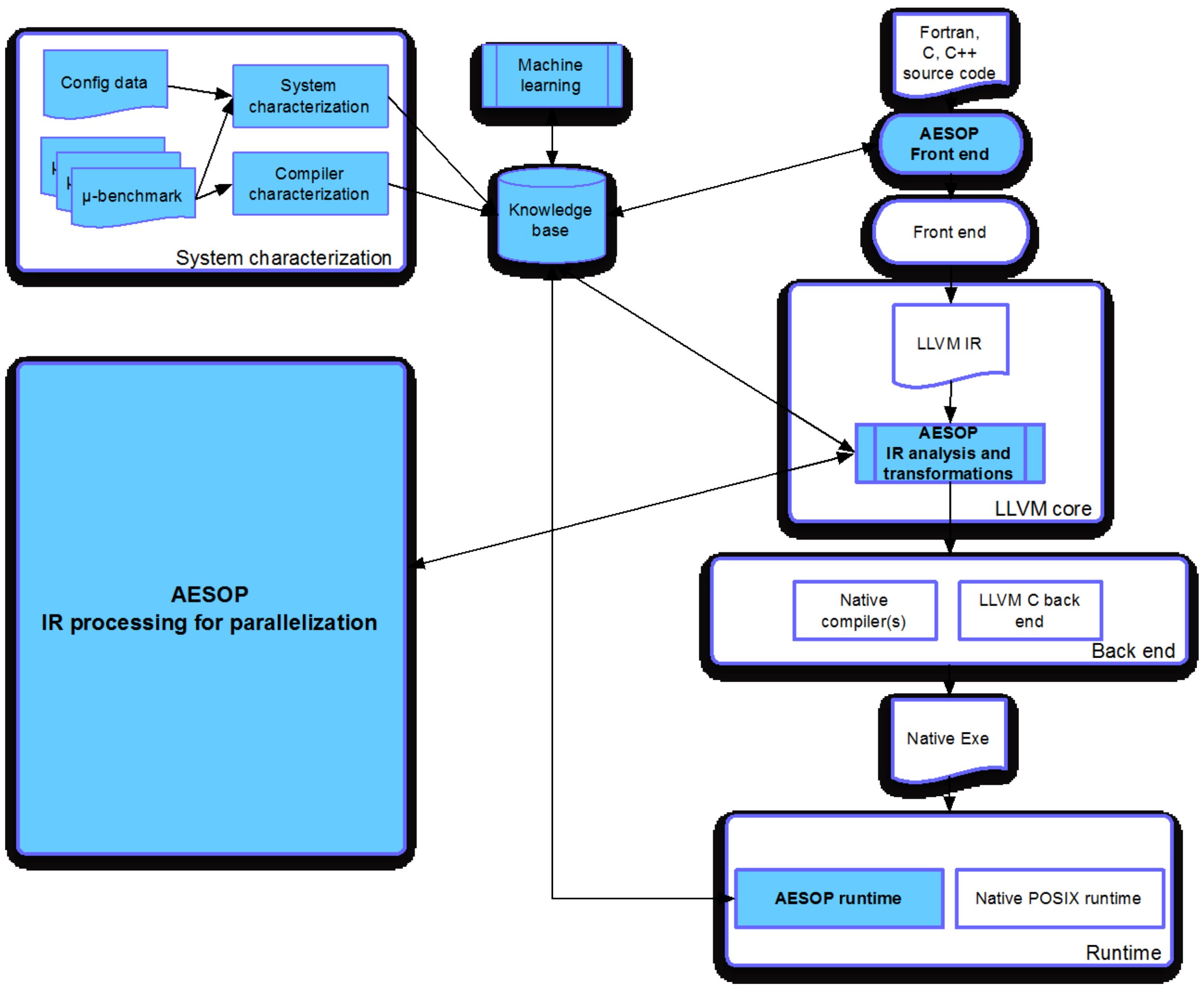


Figure 4. LLVM and the AESOP Compiler

LLVM as modified by and integrated with the full AESOP system. Stock LLVM components are the white boxes and the AESOP components are the blue boxes.

3.1.2 Other (non-open-source) research tools utilized

Even when open source was not available, our teams utilized existing successful research tools wherever possible. They were adapted as necessary for our situation. In the characterization area four such tools were utilized:

X-ray for automatic measurement of hardware parameters (Yotov 2004).

Active Harmony for tuning parallel applications in parallel (Tiwari 2009).

Servet for autotuning on multicore clusters (Gonzalex-Domingues 2010).

P-ray a suite of micro-benchmarks for multi-core architectures (Duchateau 2008).

3.2 Parallel research thrusts driven by PIs at each sub-contractor.

Teams had provable expertise in their area of compiler development and began with their most recent academic results.

3.2.1 Characterization (Alan Sussman – University of Maryland)

Characterization is the automatic determination of relevant parameters about a computing system. For AESOP this includes machine characterization, operating system characterization, and application characterization. Since some of the important parameters may change over time as the system is running, some of these parameters must also be determined dynamically and not just at boot time. Any attempt to produce optimized code must be based on information about the system on which the program will be executing, even for the simplest serial codes. For automatically parallelized code, the information required is much more extensive. It is at best extremely difficult and time consuming and, at worst impossible, to obtain the required information for a new system from documentation or spec sheets; even when it can be found, it will reflect the performance of the system under ideal conditions, not what the application will encounter during actual execution.

To port a manually threaded moderately complex application from a single dual core node to four quad core nodes, with today's tools, can take on the order of weeks. Furthermore, to re-port the same application to a different multicore architecture can again take weeks to months. To solve the problem of characterizing and then targeting multicore systems, we created a set of micro-benchmarks based on the X-Ray, Servet and P-ray tools, with parameters that are systematically searched by our continuous optimization environment directed by our knowledge-based search engine, to automate the process of providing the compiler with the information it needs to auto-parallelize. The impact is that we have succeeded in creating a simple, automated compiler configuration process for multicore systems.

New publications resulting from this work:

(Sussman, Lo and Anderson 2011) (Teodoro and Sussman 2011).

3.2.2 Affine parallelizer (Rajeev Barua – University of Maryland)

A particular sub-class of program fragments for which a greater degree of optimization can be achieved compared to general-purpose compilation methods is affine array references in loops. A reference to an array in a loop is said to be affine if its array indices are linear (affine) combinations of enclosing loop induction variables. Affine references are very important because many scientific and multi-media programs use them so heavily.

Figure 5 shows the flow diagram of the affine parallelizer. The diagram only shows the affine parallelizer in AESOP without reference to how it integrates with other parallelizers, but that is done to keep the figure simple and focused on the material relevant for affine code.

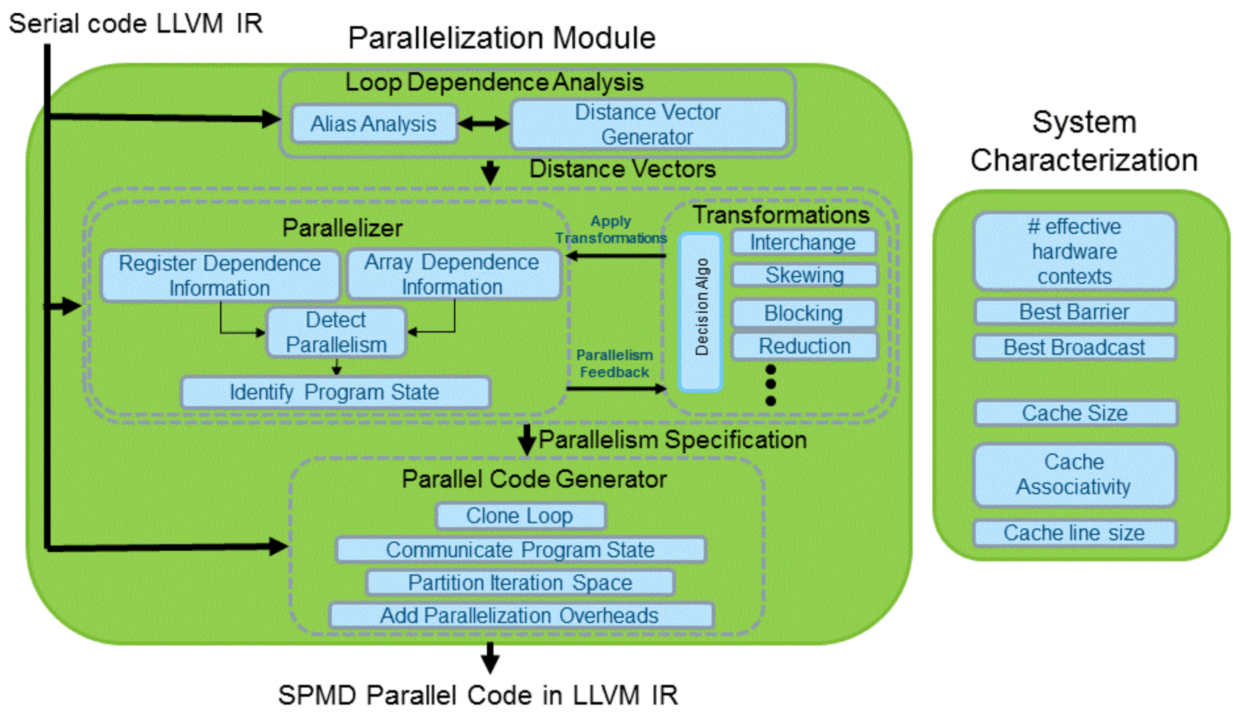


Figure 5. Affine Parallelization Flow Diagram

Diagram showing the major subsystems of the Affine Parallelizer and how they interact with each other and utilize various data values created by the system characterization system.

Given the maturity of prior work on affine parallelization, the similarity of the flow diagram to prior work is to be expected. Initially the serial intermediate representation (IR) of the input program is fed to loop dependence analysis, which uses the method described later in this chapter to compute distance vectors (Allen and Kennedy 2002) (U. Banerjee, Speedup of Ordinary Programs 1979) (M. Wolfe 1989). Dependence analysis is enhanced by alias analysis, which aims to prove that different memory references, both affine and non-affine, in the loop do not alias, and hence cannot have a loop-carried dependence. Distance vectors thus generated are passed to the affine parallelizer.

The affine parallelizer detects which loop dimensions are parallel using dependence information in the standard way. However, the effectiveness of affine parallelization can be enhanced by affine transformations (M. E. Wolf 1991) (Bacon 1994) (McKinley 1998) (Bondhugula 2008) (Bastoul 2004) such as loop interchange, loop skewing, loop reversal, loop fusion, loop fission, reductions, array privatization, blocking and strip-mining. These transformations are well known to improve affine parallelism by making the parallelism more coarse-grained, by improving cache performance, or by exposing more utilizable parallelism. The precise role of each transformation, as well as the decision algorithm used to select which transformations to apply, and in what order, will be discussed in section 4.5.

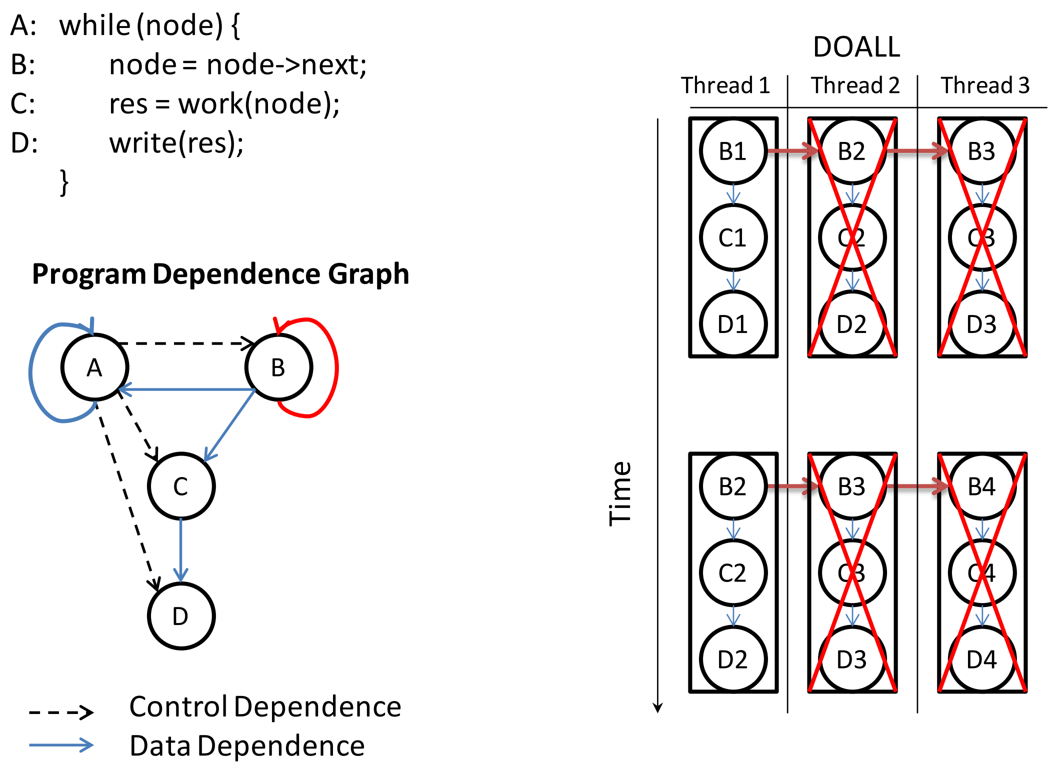
Once the transformations are complete, the code generation is done. As a first step, loop dimensions to be converted to parallel are cloned to a new form in which the iterations are to be divided among cores. The ranges of iterations are computed in the Partition Iteration Space module. The code to communicate needed shared variables in the loop from the main thread to the parallel threads is inserted in the Communicate Program State pass. Finally the synchronization in the form of broadcasts and barriers is added in the Add Parallelization Overheads pass.

New publications resulting from this work: (Yellareddy, et al. 2011)

3.2.3 Non-affine parallelizer (David August – Princeton University)

The most straightforward type of parallelism that can be extracted from loops is Independent Multi-Threading (IMT). IMT does not allow cross-thread dependencies and was first introduced to parallelize array-based scientific programs. The most successful IMT technique, DOALL parallelization, satisfies the condition of no inter-thread dependencies by executing loop iterations that it can prove have no inter-iteration dependencies in separate threads. A primary benefit of IMT parallelism is that it scales linearly with the number of available threads. In particular, this scalability has led to the integration of DOALL into several compilers that target array-based, numerical programs.

IMT's requirement that there are no cross-thread dependencies within a loop is both a source of scalable parallelism and limited applicability. While IMT techniques have enjoyed broad applicability for scientific applications, they have not been used to parallelize general-purpose applications, which have many inter-iteration dependencies.



```
A:  while (node) {
B:        node = node->next;
C:        res = work(node);
D:        write(res);
    }
```

Figure 6. DOALL Execution of Simple Loop

Example loop, program dependence graph and DOALL execution.

Figure 6 shows an example loop and the corresponding program dependence graph. All of the nodes are dependent on node A, the while statement controlling the loop. This is a rare dependence; in most iterations, the loop will continue to execute rather than terminate, so dependencies from node A may be speculatively ignored. However, node B also has an inter-iteration dependence to itself (shown in red) that cannot be speculated with low misspeculation rate. Thus, DOALL parallelization is ineffective, as shown on the right: no iterations are able to execute in parallel, and execution is effectively serialized.

Cyclic Multi-Threading (CMT) techniques, which allow cross-thread dependencies to exist, have been developed to solve the problem of applicability. To maintain correct execution, dependencies between threads are synchronized so that each thread executes with the proper values. Though initially introduced to handle array-based programs with intricate access patterns, CMT techniques can also be used to extract parallelism from general-purpose applications with complex dependence patterns.

The most common CMT transformation is DOACROSS, which achieves parallelism in the same way as DOALL, by executing iterations in parallel in separate threads. To maintain correct execution, the inter-iteration dependencies that cross between threads must be respected. These dependencies are synchronized to ensure that later iterations receive the correct values from earlier iterations.

Unlike IMT transformations, CMT transformations do not always experience linearly increasing performance as more threads are added to the system. In particular, when the threads begin completing work faster than dependencies can be communicated, the critical path of a DOACROSS parallelization becomes dependent upon the communication latency between threads. Because of this, DOACROSS parallelizations are fundamentally limited in speedup by the size of the longest cross-iteration dependence cycle and the communication latency between processor cores.

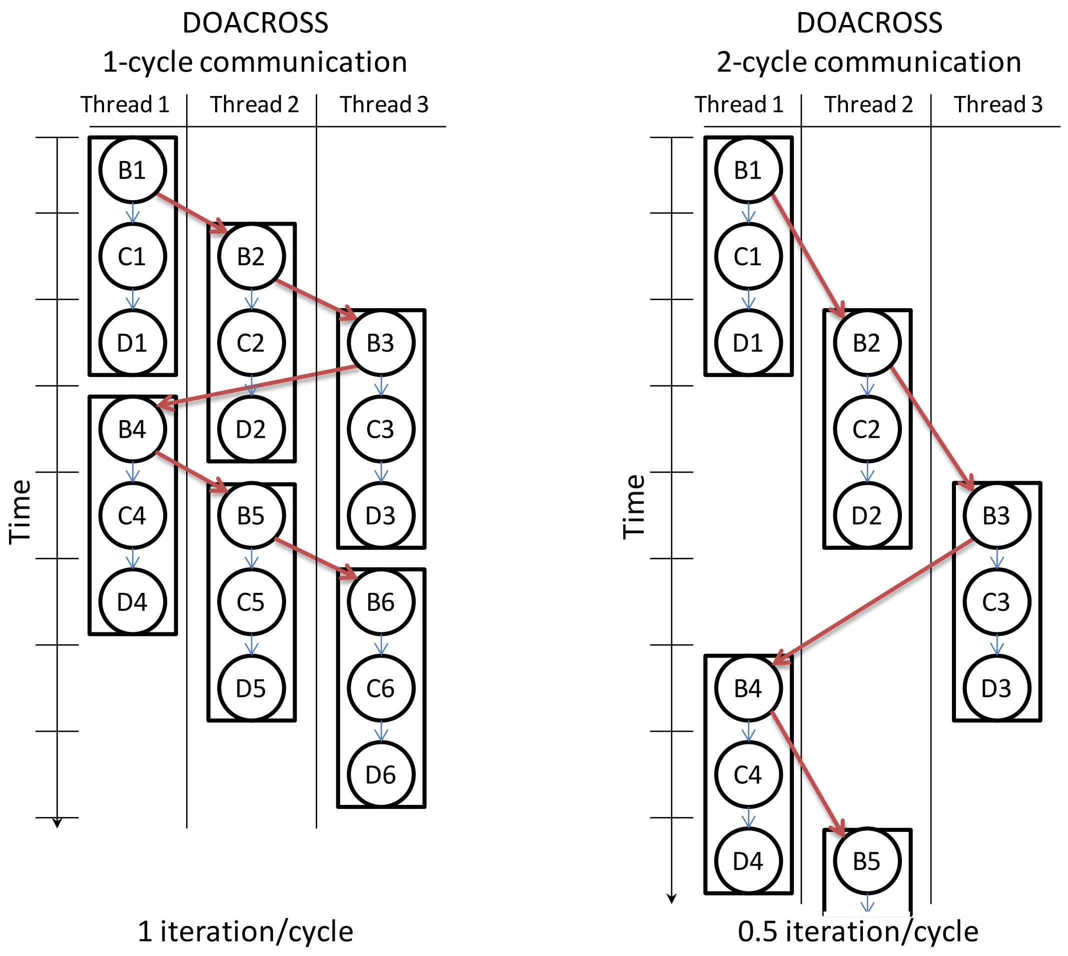


Figure 7. DOACROSS Parallelization

DOACROSS parallelization with varying communication latency.

The left half of Figure 7 shows that DOACROSS is able to respect the red inter-iteration dependence using cross-thread communication, achieving parallelism. However, the resulting cyclic communication patterns place cross-thread communication on the critical path. As shown on the right half of the figure, when communication latency increases from 1 cycle to 2 cycles, throughput decreases by half.

There are often many inter-iteration, cross-thread dependencies in general-purpose programs, which leaves little code to execute in parallel. Synchronizing these dependencies only increases the number of threads where communication latency will become the limiting factor. Additionally, variability in execution time in one iteration can potentially cause stalls in every other thread in the system, further adding to the critical path and limiting the attainable speedup.

In order to achieve broadly applicable parallelizations that are not limited by communication latency, a parallelization paradigm is needed that does not spread the critical path of a loop across multiple threads. Pipelined Multi-Threading (PMT) techniques have been proposed that allow cross-thread dependencies while ensuring that the dependencies between threads flow in only one direction. This ensures that the dependence recurrences in a loop remain local to a thread, ensuring that communication latency does not become a bottleneck. This paradigm also facilitates a decoupling between earlier threads and later threads that can be used to tolerate variable latency. The most common PMT transformation is Decoupled Software Pipelining (DSWP), an extension of the much earlier DOPIPE technique, allowing it to handle arbitrary control flow, complicated memory accesses, and all other features of modern software.

As in CMT techniques, DSWP synchronizes dependencies between threads to ensure correct execution. However, instead of executing each iteration in a different thread, DSWP executes portions of the static loop body in parallel, with dependencies flowing only in a single direction, forming a pipeline.

DSWP ensures that all operations in a dependence cycle execute in the same thread, removing communication latency from the critical path. This has the additional benefit of ensuring that dependencies flow in only one direction, which allows DSWP to use communication queues to decouple execution of different stages of the pipeline, enabling tolerance of variable latency.

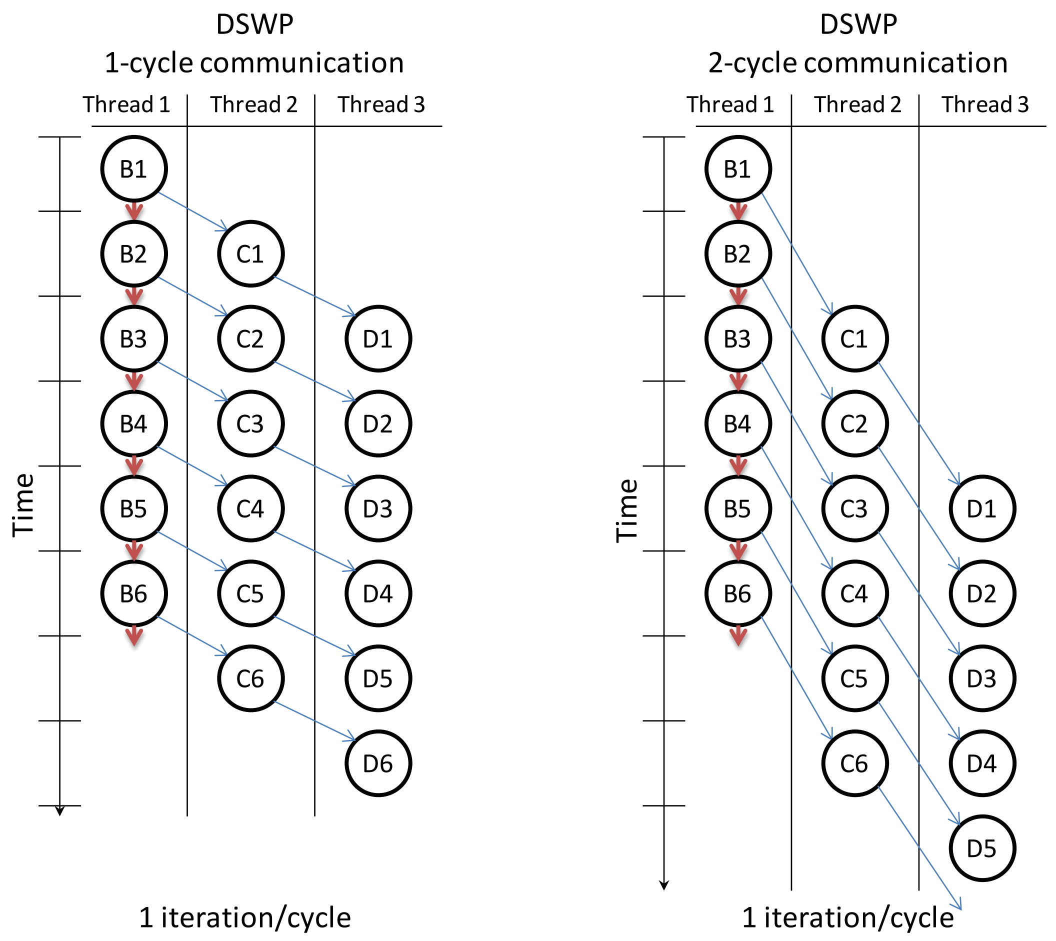


Figure 8. DSWP Breaks Iterations Across Threads

DSWP breaks iterations across threads to keep cyclic communications local.

Figure 8 shows the execution of the application partitioned with DSWP. Cyclic dependencies are kept local, and communication between threads flows in a single direction. Thus, when communication latency doubles, the throughput of the loop remains the same.

The speedup of a DSWP parallelization is limited by the size of the largest pipeline stage. Because DSWP extracts parallelism among the static operations of a loop and must keep dependence recurrences local to a thread, the largest pipeline stage is limited by the largest set of intersecting dependence cycles in the static code. In practice, this means that sometimes DSWP can scale to only a few threads before performance no longer increases with additional threads.

However, by isolating dependence cycles into pipeline stages, DSWP actually creates new opportunities for scalable parallelism. Large stages tend to be the result of inner loops whose static code cannot be split among multiple pipeline stages. However, a pipeline stage that contains no outer loop-carried dependence can execute multiple outer loop iterations in parallel. The Parallel-Stage DSWP (PS-DSWP) extension to DSWP uses this insight to achieve data-level parallelism for such stages, essentially extracting IMT parallelism for that stage. DSWP and PS-DSWP work together, isolating loop-carried dependencies in their own stages to allow the bulk of the computation to execute in parallel stages.

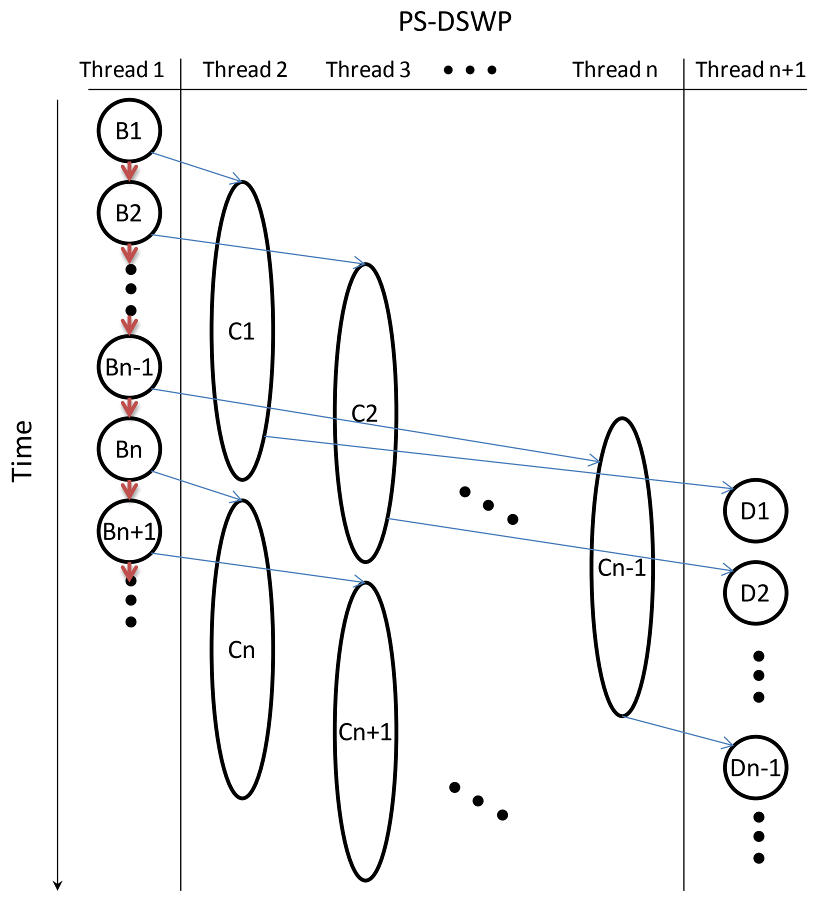


Figure 9. PS-DSWP Replicates Large Pipeline Stages

PS-DSWP replicates large pipeline stages to achieve scalable parallelism.

Figure 9 shows the execution of the example application where stage C (corresponding to the work statement in the C code) has been replicated across a large number of threads. Because it handles inter-iteration dependencies, keeps cyclic dependencies local, and replicates parallel stages, pipelined parallelization is generally applicable, latency tolerant, and scalable. Much more detail on how AESOP's non-affine (software pipelining) parallelizer works is described in section 4.6.

New publications resulting from this work: (August, et al. 2011) (Raman, et al. 2010) (Jablin, et al. June 2011)

3.3 Divide-and-conquer then re-unite and integrate.

Initially each AESOP sub-team focused on a specific subsystem and for the first year worked semi-independently until they had some identifiable results. At that time, dependencies between subsystems required integration.

The system characterization team was able to work most effectively independently running tests on various machines extracting the static characterization parameters and checking against hand tests as well as working with the evaluation teams on accuracy and methodology.

The affine team was able to develop a specialized parallelizer code base completely independent of any other group and test it using specially constructed loops that exercised various aspects of their system.

The non-affine or software pipelining team similarly worked independently and developed first their crucial core data structures such as the PDG, then perfected their high performance inter-thread communication queue code and finally developed the core of the parallelizer before integration was necessary. For testing purposes they read in static system characterization parameters.

When integration began after approximately the first year of the contract was complete, it was the new negotiation framework that formed the basis of that integration (described in section 4.3).

Throughout, the development was iterative so that initial results would inform subsequent development. We believe this "divide-and-conquer" initially allowed each team to perform research, move forward and retreat and overall to make the fastest forward progress possible before dependencies on other teams slowed them down. Similarly, we believe after a certain point, integration is essential especially to avoid duplication of shared data structure development and to begin to exercise the critical negotiation framework components.

3.4 A "spike" approach to integration was used

Spikes are successively more extensive realistic test applications that exercise the maximum extent of the system, albeit in as simple way as possible for that stage. In the terminology of agile software development, a spike is a "story" that represents a particular requirement of the compiler that must be broken down into a small enough next step that it can reasonably be accomplished in the next iteration of development. This technique has been used especially well on compilers for 20 years. The idea is to build the system to be functional but limited end-to-end and then to flesh out each subsystem as the spikes exercise more and more of the compiler.

A good way to think about this harkens back to the very first C++ compiler developed at Borland International. Spike 1 was the simplest C++ program that would compile and then execute and print out “Hello world” on standard out. That meant the parser, analyzer, assembly code generator, assembler, linker and loader all worked at least minimally. No other program would yet compile but end-to-end all the basic pieces were already in place. Subsequent spikes were more and more complex and required that each of the stages in the compiler were more capable and fleshed out. We used the same approach in AESOP.

Due to the cancellation of the AACE program only three spikes were completed. The first spike had separate simple affine and non-affine code that only required a minimal set of transforms. The subsequent spikes added more complexity to each piece of code and then mixed affine and non-affine together to test the ability of the negotiator to make the correct assignments and to break dependencies appropriately.

3.5 Leveraged work of other AACErs to the maximum extent

Characterization worked with the T2 teams to develop common methodologies for running and measuring tests. Tests were run repeatedly until results converged. Then blind runs were completed and the T2 teams published detailed evaluations of each performer which were also used to improve our characterization system accuracy.

4.0 Results and Discussion

4.1 System Characterizer

As part of installing the AESOP compiler, it is necessary to obtain the initial characterization of the machine that the compiler will be generating code for. This is static data gathered typically at compiler install time. In many cases, this will not be the machine that the compiler is running on; some systems require cross compilation for access to at least some of their resources. Even on a machine like the Dash system at San Diego Supercomputer Center (SDSC), compilation would likely happen on a single compute node, where execution might happen on the 128-core, 16 compute node, supernode.

The basic framework for the install-time characterization process is provided by the X-Ray system (Yotov 2004). X-Ray itself has four pieces, which have been modified and substantially enhanced to support the AESOP project:

- A utility library, which must build on every target platform.
- A set of small libraries, each of which provides some needed functionality: timing, synchronization, management of thread affinity, and so on. Each library can have several different implementations; on every target platform, at least one implementation of each library must be able to build.
- A framework for a set of "nano"-benchmarks, actually generated and run at characterization time, that characterize very low-level operations: the execution time of a sequence of 64-bit floating point adds is an example. On a system requiring cross-compilation, these must be compiled for, and executed on, the compute node.
- A set of "platform" benchmarks. Unlike nano-benchmarks, these run the same code on every system, but they also must be compiled for and executed on the compute node.
- A set of "micro"-benchmarks , which generally use the output of a sequence of platform or nano-benchmarks to characterize a specific aspect of the system. For example, a nano-benchmark might be used to obtain the cost of a particular sequence of memory accesses; a micro-benchmark could then use many such runs, with different parameters, to measure the size of the L1 cache. Micro-benchmarks need not run on the compute node.

4.1.1 Instruction performance

It is important for the compiler to be able to estimate the cost of a particular code sequence. This is a complex problem, and any answer that system characterization can provide based on benchmarks is at best a rough approximation. Such measurements cannot take into account the state of the cache at all levels, nor the effects of whatever pipelining, instruction scheduling, and so on is happening within the processor, beyond the compiler’s control.

What the benchmarks can provide is information about the average latency of a single instruction of a particular class, e.g., 64-bit floating-point division, and about the average throughput, that is, how many such operations can complete in a single cycle. Generally speaking, if there is no pipelining, the throughput is the inverse of the latency: a latency of 2 (integer additions; the standard unit of measurement here is the time required for a single 32-bit integer addition) implies a throughput of 0.5. If the floating point unit can have several adds in flight at one time, and can ingest a new one every cycle, the result might be a latency of 5, and a throughput of 1.

The approach taken here is an adaptation of X-Ray. For each class of instruction being evaluated, the basic measurement is throughput with varying constraints. This is measured by code generated based on a template, with variations depending on the operation being measured, and the type and size of the operands. The generated code must remain machine independent, since the X-Ray based tool does not know the instruction set architecture of the target machine, and must also prevent the compiler from realizing that in fact the benchmark is performing no useful computation, so could just be compiled as an empty function. For example, Listing 1 is a part of one of the generated nano-benchmarks for determining the throughput of integer multiplication.

Listing 1
A sample nano-benchmark

```
register long i = R;
  volatile long iv = 0;

// The r_p array is declared volatile. It is full of
// constants, which especially for floating multiply and
// divide must be carefully chosen to avoid error values
// (NaN, +Inf) and identifiable shortcuts, but the
// compiler should not treat them as constants.

  register I32 r0 = LOAD__I32(r_p + 0);
  register I32 r1 = LOAD__I32(r_p + 1);
  register I32 r2 = LOAD__I32(r_p + 2);
  register I32 r3 = LOAD__I32(r_p + 3);
  register I32 r4 = LOAD__I32(r_p + 4);
  register I32 r5 = LOAD__I32(r_p + 5);
  register I32 r6 = LOAD__I32(r_p + 6);

  switch (iv)
  {

// The case statement prevents the compiler from combining
// operations, as it otherwise might.

  case 0:
  loop:
```

Figure 10. Listing 1: A Sample Nano-Benchmark

```
// Reloading the “registers” here keeps multiply and
// divide from overflow or underflow.

  r0 = LOAD__I32(r_p + 0);
  r1 = LOAD__I32(r_p + 1);
  r2 = LOAD__I32(r_p + 2);
  r3 = LOAD__I32(r_p + 3);
  r4 = LOAD__I32(r_p + 4);
  r5 = LOAD__I32(r_p + 5);
  r6 = LOAD__I32(r_p + 6);

// This consists of several sequences of six multiplies.
// There are no dependencies within each sequence, but the
// corresponding elements of the next sequence depend on
// the first one.
// Thus all six elements of the sequence could be in
// process at the same time, but the first element of the
// second sequence has to wait for the first operation in
// the first sequence to complete.
// The GENERATE macro expands, roughly, into r0 *= r6.

  case  1:    GENERATE(MULTIPLY,I32,r0,r6);
              GENERATE(MULTIPLY,I32,r1,r6); ;
  case  2:    GENERATE(MULTIPLY,I32,r2,r6);
              GENERATE(MULTIPLY,I32,r3,r6); ;
  case  3:    GENERATE(MULTIPLY,I32,r4,r6);
              GENERATE(MULTIPLY,I32,r5,r6); ;

// The second sequence starts here.

  case  4:    GENERATE(MULTIPLY,I32,r0,r6);
              GENERATE(MULTIPLY,I32,r1,r6); ;
  case  5:    GENERATE(MULTIPLY,I32,r2,r6);
              GENERATE(MULTIPLY,I32,r3,r6); ;
  case  6:    GENERATE(MULTIPLY,I32,r4,r6);
              GENERATE(MULTIPLY,I32,r5,r6); ;
  …
  case 258:   GENERATE(MULTIPLY,I32,r4,r6);
              GENERATE(MULTIPLY,I32,r5,r6); ;

  again:
        if (--i)
              goto loop;
  }

  if (!iv)
  {
        return;
  }

// Storing the results here ensures that the compiler
// executes the loop; without this, some optimizers will
// notice a long sequence of irrelevant computations, and
// discard the whole thing.

  STORE__I32(r_p_s, r0);
  STORE__I32(r_p_s, r1);
  STORE__I32(r_p_s, r2);
  STORE__I32(r_p_s, r3);
  STORE__I32(r_p_s, r4);
  STORE__I32(r_p_s, r5);
  STORE__I32(r_p_s, r6);

  goto again;
```

Figure 11. Listing 1: A Sample Nano-Benchmark (continued)

4.1.2 Cache size and characteristics

Modern systems usually have one or more levels of cache, of various types: data, instruction, both, and translation lookaside buffer (TLB). The characteristics of the instruction cache can be important in generating code, especially loops: if a given loop fits entirely in the instruction cache, it can run much faster than one that has to hit more remote caches, or even main memory, each time the loop executes.

The AESOP compiler optimizations are currently only concerned with data cache parameters. We've found that the Servet system (Gonzalex-Domingues 2010), as modified, generally does a better job of measuring cache sizes than X-ray, but it does not report either associativity or block size. Our approach is to run Servet first to determine the sizes of all the caches. We include the L3 cache in this measurement, if it exists, not because the AESOP compiler needs that number, but because we use it in the benchmark for measuring Unified Memory Access (UMA) parameters, discussed below.

Roughly speaking, X-ray makes its decision about cache size based on the first derivative of a curve showing average access time for a given presumed cache size. Servet obtains equal or better results in most cases, more quickly, by looking at the second derivative. It also deals with an issue that is common to level 2 (L2) and higher caches. Typically the L1 cache is accessed using virtual addresses — that is, before the address has passed through the memory management unit (MMU). L2 and higher-level caches are typically shared — between processes, or most often on modern systems between contexts on the same chip. Consequently, they must be accessed using physical addresses as produced by the MMU. But it is difficult, often impossible, for user-level code to ensure that two pages with consecutive logical addresses also have consecutive physical addresses, making it correspondingly difficult to obtain the precise control over memory access patterns that is required for the benchmarks measuring cache parameters to be highly accurate. Some systems provide “huge” pages, which are guaranteed to be physically contiguous, but the interfaces to obtain those super-pages are not part of the POSIX standard even when they are available; other systems use “page coloring”, where allocations spanning many pages are given the same color, and kept together in physical memory where possible. Servet is able to exploit page coloring, while the X-Ray based tool does not. Our modifications in Servet's approach use dynamic thresholds to find the peaks where the likely cache boundaries are.

The X-ray cache micro-benchmarks determine the associativity and block size of the L1 and L2 caches by measuring the average time per memory access, using code that accesses the elements of carefully selected sets of memory addresses in a particular order. Each set of memory addresses is generated by a separate nano-benchmark, and the nano-benchmarks are aggregated using the X-ray based tool's internal mechanisms into several micro-benchmarks to run on the machine that is to be characterized.

4.1.3 Contexts

The operating system typically will provide one or more ways to find out how many contexts (cores, or “threads”) a given computer system provides or supports; sometimes it may also report the number configured and the number actually online. However, there is no portable query to determine other factors that may be of interest when parallelizing code. For example, some processors with hyperthreading may support two execution threads per physical core, but the physical core may provide only one floating-point unit. In such a case, one would rather avoid the placement of two floating-point intensive threads on contexts supported by the same core. Similarly, threads that are doing many memory accesses that leave the chip (either to the L3 cache or to main memory) may end up overloading the chip's memory interface, and might be better off executing on different chips, or with less parallelism. In addition, some contexts may be permanently occupied by high-priority tasks; the system will report them, but they are in effect not available to the compiler. Only a benchmark can, in a portable way, determine these system characteristics.

Benchmarks are used to determine how many effective contexts are available for integer intensive computation, floating point intensive computation, and memory intensive computation. In each case, the basic principle is the same: run a platform benchmark that provides a particular kind of load on more and more contexts until performance drops off; that is until execution time of an individual nano-benchmark increases.

4.1.4 Pairwise communication costs

Both for software pipelining, which uses a point-to-point queue to pass data between threads, and for modeling the cost of software barriers, it is critical to have an accurate measure of communication costs among various contexts in a system. Depending on where the contexts are in relation to each other, the amount of data being exchanged, the rate at which data is moved, and the exact access pattern, the data may:

- pass through the L1 cache (for example on a system with hyperthreading, where the threads are on the same physical core);
- pass through a shared L2 cache;
- pass through the L3 cache;
- pass through local main memory; or,
- pass through distributed shared memory on a Non-Uniform Memory Access (NUMA) system like Penguin.

A common access pattern in parallelized code is a FIFO queue, where data is written into a queue in shared memory by one thread, and consumed by another. With care, this can be implemented with very little contention, and with enough information about cache size and sharing, can be kept in cache at some level. Some parallel threads may be able to tolerate a high degree of latency on the communication path, even at the level observed between nodes on Penguin, but the compiler needs to know what the latency is.

4.1.5 Barrier modeling

Especially for affine parallelization transforms, the use of barriers to synchronize threads at the end of a parallel section is critical. The cost of those barriers can be a significant factor in deciding whether to parallelize a loop, and, if so, how many parallel threads to spawn.

The performance of a barrier is determined by the number of threads being synchronized, by the assignment of those threads to contexts on a particular system, and by the barrier implementation itself. For example, for a small number of threads able to access a single location through a shared cache, a central barrier, implemented simply as a counter, may be most efficient. On the same system, with the same number of threads, a dissemination barrier may be better when the threads are dispersed among multiple nodes.

It is often too computationally expensive to directly measure the cost for every possible use of a barrier. On the Penguin system, with 16 four-context nodes, we can assume that a four-thread barrier with all four threads on the same node will have consistent behavior whichever node it is used on. If, on the other hand, the four threads are split between two nodes, the barrier cost will be affected by the exact choice of nodes: the cost between nodes 0 and 1 in the system is not the same as the cost between nodes 0 and 15. The number of combinations even for this simple case is in the millions; obviously equivalences can be found to reduce this significantly, but the same issue will arise for five threads, six threads, and so on.

In a more general approach, each type of barrier provided by the runtime library has a manually constructed cost function. The function accepts as its input the set of contexts being used, as well as the static characterization of the various cache levels and pairwise communication costs, and returns an estimate of the cost of passing through the barrier. Executing the cost function for each barrier implementation in turn allows one to choose the best barrier for a given case.

The construction of the cost function is not automatic. It requires careful analysis of the exact implementation, rather than just of the basic algorithm, and a mapping of the sequence of memory accesses to available communication and memory benchmarks.

4.1.6 Dynamic System Characterization

The static characterization produced at compiler installation time may be lacking in several possible ways:

- The hardware configuration of the system may have changed in some way. A processor went off line, or came back on line, for example.
- The software configuration may have changed. A high-priority process was added, or completed, or there is just an unusually heavy load on the system at the moment.
- The characterization of instruction latency and throughput can only allow rough approximations of the time required to execute a particular sequence of instructions. The benchmarks can measure the throughput of a sequence of 64-bit floating-point divides with no intervening code, but that is not a common computation.

Dynamic characterization allows more information about the system to be gained either during compilation, or during program execution. At compile time, the primary value of dynamic characterization is more accurate measurement of the cost of an instruction sequence; instead of a sequence of divides, the compiler can provide a block of code from the application to the characterization tool, with some idea of the data that it might be operating on. This can be difficult to execute in a cross-compilation environment, such as some Crays and embedded systems, but where available it can help produce better code. All dynamic characterization usage scenarios will require having the compiler insert instrumentation into the code it produces, to perform measurements of various code behaviors (e.g., time for a code section, L1 cache miss count, etc.). The measurements to be performed will be closely tied to the compiler optimization techniques that may be applied to a given code sequence.

At run time, characterization can be used in several ways that need not directly involve the compiler. An obvious characterization task is to measure the actual time required to execute a particular sequence of code on a particular data set. This can be fed back into the AESOP knowledge base for use in future compilations, and at run time can be used to select among alternate implementations.

Characterization at compiler install time must be system independent, but it may be possible for the run time environment to take advantage of more system-specific information to provide a better idea of how many contexts are available. Even a system independent test can quickly determine how much contention there is for the existing contexts, and whether the number of contexts has changed since static characterization was last run.

It is not until run time that the compiled code will know exactly which contexts are available. One example is where the first choice made by the compiler assumed the availability of four contexts on the same chip, yet at run time the only four contexts that are available are split between two chips. This could lead to the selection of an alternate version of the compiled code, or when fed back through the knowledge base, might cause the compiler to generate an alternate version in future compilations of similar code.

Overall, the X-Ray based approach to static system characterization has proved to be sound, enabling us to build micro-benchmarks that can accurately measure the desired system characteristics. Design of the individual micro-benchmarks, and associated nano-benchmarks, is still challenging, especially in the context of fully automated characterization. A main weakness of the original X-Ray tool was in selecting parameter values from measurements that do not show large changes in performance (e.g., given a set of L1 cache sizes and associated benchmark timings, at what cache size does performance of the benchmark decrease enough to say that the data reference pattern is causing cache misses). We have therefore augmented the X-Ray based tool with methods from the Servet project, including better ways of detecting performance breakpoints, which has greatly improved the accuracy of our system characterization tool. And while we have a design for utilizing dynamic characterization information, much work remains to be done on using measurements from instrumented code to make optimization decisions, either at run time or across compilations.

4.2 Critical data structures

Control Flow Graph, Program Description Graph and Liberty Queues are the three most central data structures used by AESOP's parallelizers. The following sections describe these central and critical data structures.

4.2.1 The Control Flow Graph (CFG)

The LLVM Compiler Framework includes an implementation of the CFG with instructions in SSA form. We will only summarize LLVM's contributions here; for extensive details, please see the documentation from the LLVM project.

In LLVM, the program is organized into functions with a single entry point, and one or more exit points. Functions are divided into basic blocks (often referred to simply as blocks), and one distinguished basic block is marked as the entry block of the function. Basic blocks contain instructions.

A basic block is a sequence of instructions with a single entry point and a single exit point. The single exit point constraint means that basic blocks contain no internal control flow. A consequence is that the last instruction in a basic block is always a terminator instruction, such as a conditional branch or return instruction. Basic blocks emphasize that all their instructions are control-equivalent (i.e. if any of those instructions execute, then all execute). For this reason, basic blocks reduce the effective size of many analyses. For instance, many data flow analyses can summarize the net effect of a basic block with transfer functions and treat each basic block as if it were a single large instruction.

The CFG of a function

The CFG of a function is the directed graph where every basic block is a vertex, and edges are drawn according to the control flow relationship. Two blocks b_i and $b_j$are adjacent iff control may flow immediately from the last instruction of b_i to the first instruction of b_j. For instance, if block b_i ends in a conditional branch to either b_i or b_k, then the CFG will feature two edges (b_i, b_j) and (b_i, b_k). One block is distinguished as the entry block of the function, and may have no ingress control flow edges. One or more blocks are exit blocks of the function, and may have no egress control flow edges.

Figure 12. The CFG of a Function

4.2.2 The Program Dependence Graph (PDG)

The AESOP team uses the PDG extensively. This choice was motivated by the PDG's emphasis on dependencies and parallelism. Instead of emphasizing the flow of control or data, the PDG emphasizes dependencies among the instructions. Starting with the premise that most instructions do not affect most other instructions, the PDG draws edges whenever this premise is violated. Specifically, the PDG uses control dependence edges if one instruction may promote/inhibit the execution of another instruction (e.g. conditional branches or return instructions), and data dependence dependencies edges whenever one instruction defines a value that another uses. Data dependence edges are further subdivided for pragmatic reasons into register and memory dependencies. Register dependencies represent exact data dependence that can be observed via the static structure of instructions. Memory dependencies capture data flow through memory, and are often as approximate as the underlying pointer analysis. Various externally observable side effects, such as I/O, are often modeled as memory dependencies.

Any two non-adjacent vertices in the program dependence graph are (by construction) independent, and thus may be executed in parallel without affecting program semantics. Conversely, adjacent vertices may be executed on separate computational units, provided that their dependence is somehow respected, perhaps by a primitive operation on an inter-core queue. These properties of the PDG have enabled us to prove correctness of several of our algorithms, such as Multi Threaded Code Generation (MTCG).

However, the PDG has other properties that make it ideal for this work. Since a lack of dependence frequently implies easy parallelism, and a dependence frequently implies expensive parallelism, it is only a short jump to conceptualize PDG edges as “bad”, or something to be minimized. Indeed, when we consider popular parallelizing transforms, it is apparent that PDG edges limit, but do not enable parallelism.

Our implementation of a program dependence graph is a write-once, read-often design. PDG objects may only be constructed by a special class known as a **`PDGBuffer`**. After construction, the PDG object is mostly immutable. This trade-off was chosen so that many similar/related PDG objects could be simultaneously live while remaining within our memory budget.

In the typical use case, the user does not need to construct a PDG object using the **`PDGBuffer`**. Instead, the user will usually request a PDG from the **`PDGBuilderPass`**, which implements the **`llvm::FunctionPass`** interface. The **`PDGBuilderPass`** is used to extend the lifetime of a PDG object across multiple passes. The lifetime is discussed in the next section.

A PDG object is a collection of vertices and edges. Vertices generally represent instructions within the program, though special source and sink vertices may also exist in certain types of PDGs, as described below. Edges represent dependences between vertices; different types of dependences, including control and data dependences, are represented by different types of edges.

One thing that is important to note is that, since vertices and edges may be shared by more than one PDG object at a time, much of their state is externalized and stored in the PDG object itself. For instance, the vertices’ adjacency lists are stored in the PDG object. Similarly, edge annotations may be view-specific, and so they too are stored in the PDG object. Thus, to iterate over the edges of a vertex, we call a method of the PDG, not a method of the vertex.

4.2.2.1 The lifetime of a PDG object

PDG objects are reference counted, and they persist so long as a reference is held (via **`incref()`**, **`decref()`** pairs). Most simple passes do not construct a PDG object, but instead retrieve one computed by the **`PDGBuilderPass`**. Since this pass holds references to its objects, the lifetime of such PDG objects is as long as the lifetime of the **`PDGBuilderPass`**. This means that earlier passes can apply annotations to a PDG object, and be assured that later passes will see the same PDG object (as opposed to a new PDG, recomputed).

4.2.2.2 Loop views of a PDG object

If you want to focus on a loop, you can create a loop view of a function PDG. These views differ from normal PDGs in the following ways:

- Loop PDGs have both source and sink vertices. These vertices represent everything before or after the loop. Edges from outside of the loop to a vertex within the loop are represented as edges from the source. Edges from a vertex within the loop to a vertex outside of the loop are represented as edges to the sink.

- Loop PDGs inherit the annotations from the parent, but may have different annotations from the parent.

- The edges may have different annotations in the loop view. The notion of a loop-carried dependence is specific to a single loop view, and so the loop-carried annotation is computed when the view is created.

4.2.2.3 Annotations on PDGEdge objects

Edges may be annotated with a fixed-length bit-vector. This allows us to add information to edges, yet still re-use them in several PDG objects. For instance, an edge may have some quality within one loop-view, but not within another loop-view. This can be compared to dependence vectors, which are different for each loop in a loop nest.

The AESOP compiler uses loop annotations to achieve two important tasks: edge removal and loop analysis.

PDG objects may be large, and the compiler may require that many be simultaneously live. Thus, the implementation of the PDG data structure is compressed in memory (discussed in the next section). However, compressed PDG representations are difficult to mutate. Hence, we decided to offer only limited mutation of a PDG object after construction. Once a PDG object has been constructed, you may no longer add new vertices or edges. However, you can remove edges. This is accomplished with the **`ANNOTATE_REMOVED_DEP`** annotation. If an edge is so annotated, other parts of the compiler will ignore that edge, as if it were not present.

We also use edge annotations to store loop-dependence analysis. We chose to store this as an edge annotation instead of a field of the edge object. This is because the same edge object may simultaneously exist in more than one PDG object, but the result of loop analysis is valid for one, but not all, of the PDGs that contain that edge. Edges that represent a loop-carried dependence (i.e. a dependence distance greater than zero) are annotated with the **`ANNOTATE_LOOP_CARRIED_DEP`** annotation.

4.2.2.4 Sharing among PDG objects

The Negotiator Framework (described next) may search many potential compositions of transformations. As a result, thousands of intermediate search nodes, consisting of thousands of PDG objects, may be simultaneously live. Thus, memory efficiency is an important issue. However, within the Negotiator Framework, most of these PDG objects are related, and tend to share some common substructure.

Thus, to store them efficiently, PDG objects are represented internally as deltas from their "parent" PDG. This implementation is largely invisible to the user; still, these design decisions do have some visible consequences, and it may be useful to be aware of them.

Both edges and nodes may belong to more than one PDG object. Every PDG object may have a parent PDG object. This relationship exists only to enable sharing, and has no meaning in relation to the source program. If the PDG has a parent, then some unspecified details are inherited from the parent. Specifically:

- The set of out-edges for a given vertex.
- The set of in-edges for a given vertex.
- The annotations for each edge.

Thus, if two PDGs are highly similar, very little storage will be required to represent the derived PDG, as only the differences in the above details need to be recorded. This PDG representation is well suited for any use that deals with large numbers of similar PDGs, such as the Negotiation Framework described in the next section.

4.3 Negotiator

Automatic parallelism shows great performance improvements for some applications, yet these results are not yet universal. Although transformations have been crafted for many common patterns in scientific and general-purpose applications, orchestrating these transformations and enabling transformations has proven to be a difficult problem. This work presents a refinement of iterative compilation in which the applicability and profitability guards of parallelizing transformations are used to narrow the search space of iterative compilation. Instead of examining all combinations of transformations, we consider those transformations that restructure a program as to enable or enhance later transformations.

4.3.1 Introduction

Compilers exist to save human effort, and so far they have been extraordinarily successful. However, traditional human-compiler interaction (HCI) does not meet the needs of today's programmer. In this multicore era, applications must be reworked to capitalize on the potential of parallel hardware. Automatic parallelization techniques have had great successes, but these results are not yet universal. Furthermore, compilers do not suggest ways of improving application performance, leaving programmers to search through unfamiliar architectures with only crude performance measurements to act as their guidance.

Manually re-engineering applications for multicore detracts programmer effort from the ultimate goals: application correctness and competitive feature sets. Figure 13 depicts the difference between traditional HCI and a compiler that can give guidance. In traditional HCI, the programmer must interpret performance measurements with respect to a mental model of the code base, language, compiler, OS and architecture, and then synthesize a strategy for improving performance. The fatal flaw is the expectation that the programmer has a mental model of the whole. Programmers do not (and should not) understand these details for the same reason that programmers prefer high-level languages to assembly: these details are below their cognitive radar. Alternatively, the proposed HCI delegates this work to the compiler, and charges the compiler with synthesizing a strategy for refactoring the application for parallelism. Whenever a compiler can act autonomously, it will. To meet programmer expectations (or equivalently, to guarantee that language semantics are honored), strategies that improve performance yet which require minor changes to program behavior are presented to the programmer as suggested modifications.

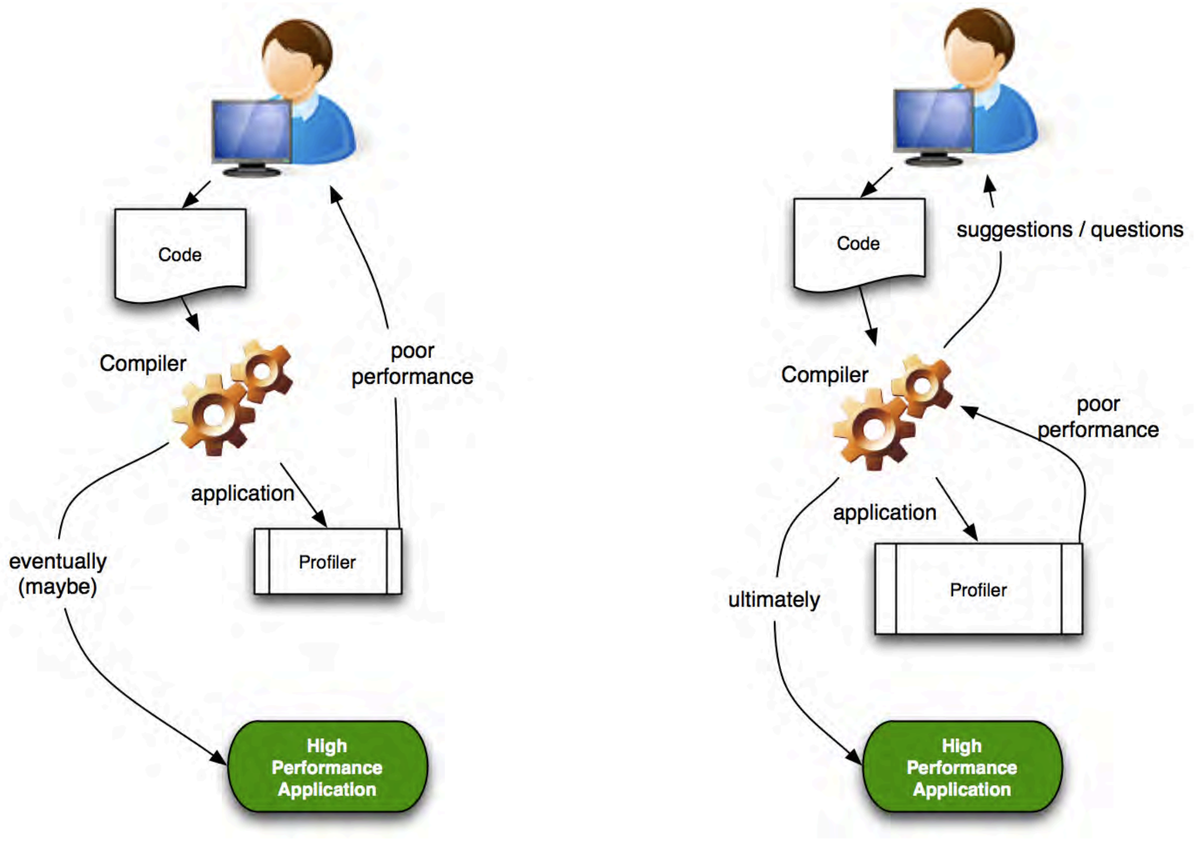


Figure 13. HCI in Negotiator

Human-computer interaction with a compiler: traditional (left) and in AESOP (right).

Automatic parallelization technologies are *almost there*. Automatic parallelization techniques are sensitive to the quality of static analysis, and suffer from the over-constrained semantics of sequential programming languages. Frequently, the difference between success and failure is specific to a small, localized aspect of the code. Advanced compiler transforms should extract parallel performance when they can, or otherwise guide the programmer towards those small changes, requiring only the minimal amount of programmer effort.

It seems as though there are two questions here: *Can a compiler re-factor an application for performance?* and, *Can a compiler guide the programmer to improved performance through meaningful feedback?* The former concerns the compiler's ability to make non-trivial, cascading changes to the source, thus requiring the compiler to comprehend the interactions among program transformations. The latter concerns the compiler's ability to formulate the inapplicability of program transformation into human-understandable terms. In actuality, these questions are the same. Through suitable abstractions, the compiler can treat the programmer as another program transformation, and focus on the interaction of program transformations. This unified philosophy greatly strengthens the system; unlike automatic transformations, the programmer-as-transformation may decide to change program semantics.

The interaction of optimizing transformations can be quite complex; one transformation may enable, disable, or drastically change the behavior of later transformations in ways that are hard to express in closed form. Although many optimizing compilers ignore this problem by following a fixed order of phases, there is no optimal static order of phases (Agakov, et al. 2006) (Cooper, et al. 2005) (Tate, et al. 2010) (Triantafyllis, Vachharajani and August 2003) (Wolf, Maydan and Chen 1996). This is referred to as the phase-order problem (Cooper, et al. 2005). However, we must solve a greater problem: not simply what order to apply transformations, but also which operating mode each should choose. Some have employed much human reasoning to study transformation interactions and distill the conclusions into phase interaction heuristics (Wolf, Maydan and Chen 1996). Others have abandoned a priori heuristics in favor of a compile-time search for good transformation sequences (Cooper, et al. 2005), but this space is exponential in the number of transformations considered. We favor the latter approach, and employ strong heuristics to let us greatly reduce the size of the search space.

On the other hand, many have studied the possibilities of compiler feedback. A natural first step is to ask the programmer specific questions about the program and use the answers (possibly in the form of added annotations) to augment static analysis (Liao, et al. 1999) (Vandierendonck, Rul and De Bosschere 2010). These are effective, but ultimately require the programmer to think in low-level terms, such as memory aliasing or the flow of values through memory. Several techniques offer guidance for loop-nest level restructuring (Liao, et al. 1999). Most intriguing, one study reports that simply reporting the inapplicability of optimization transforms gradually educated programmers how to take better advantage of their compiler (M. Wolfe 2005).

AESOP has developed the Negotiator, the first interactive compilation framework that actively exploits optimization-interaction. We phrase the problem as the search for a series of transformation steps, each classified as either a criticism or a remedy. Criticisms describe the properties of a program that prevent optimization. Remedies represent transformations which change the properties identified by criticisms, including actions potentially suggested to the user. This search is more efficient than brute force because it pursues only those paths that are likely to enable optimization.

The system allows for any pair of transformations to interact. In order to accomplish this, we created a pluggable, decoupled optimization-interaction framework. A naive system with N transformations must be aware of $O(n^2)$ pair-wise transformation interactions. The amount of human reasoning necessary to create this naive system grows too quickly to support a large number of transformations. We instead introduce the criticism identification methodology, which uses a language of criticisms to decouple transformations. Adding new transformations to the system does not require the programmer to consider their effect on other transformations, yet the overall power of the system grows as quickly as before.

This decoupled nature allows our Negotiator to give better feedback to the user. If for instance, an optimization can be enabled by either an automatic transform or by some action on the user's part with equal gains, the system will opt for the automatic transform. This has the effect of reducing the amount of feedback to the user, increasing the value of each communication. It supports partial remedies for situations in which a recommendation will partially solve a problem, possibly in conjunction with an automatic transform.

The contributions of this research are:

- We present the Negotiator, an optimization-interaction-aware interactive compilation framework that supports arbitrary interaction of enabling transformations, optimizations, and compiler feedback in an efficient manner.

- We introduce the criticism identification technique, which characterizes the interaction among transformations in terms of criticisms and remedies, allowing local reasoning about transformations within sequences. We instantiate this technique against a concrete language of program dependences, and explore efficient implementations for several transformations.

4.3.2 Practical results

We treat every transformed variant of a program as a point in program space. Semantic-preserving transformations link two points in program space, and give it structure. Finding good optimization strategies is then the problem of finding the path to the most optimal point in this space that is reachable from the input code. This space is huge. Adding new transformations to the system increases the size of the space that is reachable from the input code.

Criticisms and remedies can be used to guide a search through program optimization space. Criticisms identify points that are better than the source, without considering reachability. Remediation tries to reach those points. Criticisms serve as a heuristic that evaluates the overall benefit of long steps of the search, and thus large branches of the search space. If no transform identifies a criticism containing a given edge, then there is no likely benefit to branches of the search that try to remedy that edge.

A *strategy* represents a concrete transformation sequence, and signifies a reachable point in program space. Since the Negotiator constructs criticisms and remedies in terms of PDGs, we choose a representation of strategies built around PDGs. This work focuses on automatic parallelization, the representation of a strategy must make explicit the presence of multiple independent partitions of the input code. A strategy is represented as a sequence of transformation steps. A transformation is represented as a set of named PDGs (the results of the transformation), as well as a symbolic expression of the running time of that transformation. Those symbolic expressions may contain as free variables the result of static evaluation of the running time of any of the blocks. In practice, critics and remediators attached additional transformation-specific meta-data to represent specific operating modes, which we represent here as the description field.

The Negotiator performs depth-first search over strategy objects, each representing a path through program space. Search begins from the identity transform (i.e. no steps) of the input code. Search proceeds by repeatedly expanding the current strategy, adding adjacent strategies to the fringe if cutoff has not occurred, and accumulating the best N strategies. Search terminates when the fringe is empty.

The routine **expand** computes adjacent strategies. It runs each of the critics on each of the blocks of the current strategy. Each identified criticism is then passed to each remediator. When remediators satisfy a criticism, the composition of the current strategy and the remedy is added to the search fringe.

We impose absolute limits on search to trade completeness for decreased running time. The system accepts limits on the number of available threads, maximum search depth and absolute running time. In our experiments, we use 8 threads, maximum depth 4 ply, and no time limit.

We explored several cut-off heuristics, most of which we rejected. We did, however, find three powerful heuristics: the *optimistic limit* heuristic, the *idempotency* heuristic, and the *super-idempotency* heuristic.

The *optimistic limit* heuristic compares the current strategy to the best strategy so far. We say that a strategy has $x = M - m$ excess threads, where M is the overall thread limit of the search, and m is the number of threads used by this strategy. The optimistic limit assumes that the excess threads can be used to gain x-fold performance improvement in subsequent steps. If this x-fold improvement would not yield a better strategy than the best so far, then the Negotiator abandons this line of search.

A transformation is *idempotent* if repeatedly applying that transformation results in code that a single application can achieve. The idempotency heuristic cuts off strategies that consecutively apply idempotent transforms. *Super-idempotency* is a stronger notion— whereas idempotency applies to transforms that are adjacent in a transformation sequence, super-idempotency applies to transforms anywhere in the sequence.

To achieve interactivity with the user, one could write remediators that ask the user and conditionally produce remedies. However, experience has shown that this presents excessive feedback. Most paths will not contribute to the recommended strategy. Instead, our interactive remediators assume that the user will approve or reject those changes after the search has completed. The question is presented to the user again only if this interactive strategy is the most performant of those examined. If the user does not approve of any step, the user can select another of the reported top strategies, or can encode that those assumptions were inappropriate and re-run the search.

4.3.3 Negotiator Example

Listing 2 shows an example of a loop that can be parallelized by the negotiator using a combination of software pipelining and affine parallelization techniques.

Listing 2
Using the negotiation framework to extract parallelism: original loop

```
typedef struct _Node {
 double array[ARRAYSIZE];
 struct _Node *next;
} Node;

Node *head; /* Linked list of arrays of size ARRAYSIZE */

double work(double arg1, double arg2);

void example() {
 Node *node;
 int i;

 for (node = head; node->next != 0; node = node->next) {
  for (i = 1; i < ARRAYSIZE; i++) {
   node->next->array[i] =
    work(node->next->array[i-1], node->array[i]);
  }
 }
}
```

Figure 14. Listing 2: Using the Negotiation Framework to Extract Parallelism: Original Loop

This loop traverses a linked list of arrays and performs some computation on each element of each array. The value of each element is computed by the work function and depends on both the previous element in the same array and the corresponding element in the previous array. Thus, the loop is not amenable to a straightforward DOALL-style parallelization because of inter-iteration dependencies in both levels of the loop nest. In addition, since the outer loop iterates over a linked list, many traditional loop transformations are not applicable.

While searching for an optimal compilation strategy for this loop, the negotiator is told by the affine critic that the dependences through the linked list are preventing affine transformations from being applied. The negotiator is able to apply the DSWP transformation as a remedy; DSWP creates a two-stage pipeline in which the first stage walks the linked list and the second stage performs the computation on each list node (array), as shown in Listing 3. This isolates the linked list traversal to the first stage, leaving the second stage free of these dependences.

Listing 3
Loop transformed with DSWP

```
Queue Q[QSIZE];

void stage1() {
 Node *node;
 int Qindex = 0;

 for (node = head; node->next != 0; node = node->next) {
  enqueue(Q, Qindex, node->array); /* Enqueue current node
*/
  Qindex++;
 }

 enqueue(Q, Qindex, node->array); /* Enqueue the last node
*/
 Qindex++;

 enqueue(Q, Qindex, 0); /* Stage 1 finished */
}

void stage2() {
 int i;
 int Qindex = 1;

  while (!ready(Q, Qindex)) { }

 while (Q[Qindex]) {
  for (i = 1; i < ARRAYSIZE; i++) {
   /* Linked list accesses replaced by queue accesses */
   Q[Qindex][i] = work(Q[Qindex][i-1], Q[Qindex-1][i]);
  }
  Qindex++;
  while (!ready(Q, Qindex)) { }
 }
}
```

Figure 15. Listing 3: Loop Transformed with DSWP

In the transformed code, stage 1 walks the linked list and enqueues the array reference in each node into a communication queue. In stage 2, references to the linked list in the loop have been replaced by indexed accesses to this queue. As a result, the loop in stage 2 is now amenable to affine parallelization. In this case, a transformation such as loop skewing will expose scalable parallelism in the stage 2 loop. Listing 4 shows how this can be accomplished: the loop now iterates diagonally over the iteration space, and all iterations of the inner loop can occur in parallel. Of course, this example assumes that the work function is re-entrant, i.e., it can be called correctly in parallel. The compiler conditions for checking whether a function is re-entrant are well known and documented in the literature. Our compiler will check those conditions to ensure correct parallelization. In the final plan of execution, given n hardware contexts, one thread will execute stage 1, iterating over the linked list, and n-1 threads will execute stage 2, performing the computation.

Listing 4
Second stage after loop skewing

```
void stage2() {
 int i, j, q;
 int Qindex = 1;

 while (!ready(Q, Qindex)) { }

 for (j = 1; Q[Qindex]; j++) {
  i = min(j, ARRAYSIZE - 1);
  q = max(1, j – ARRAYSIZE + 2);
  for ( ; i > 0 && q <= Qindex; i--, q++) {
   /* These iterations can occur in parallel */
   Q[q][i] = work(Q[q][i-1], Q[q-1][i]);
  }
  Qindex++;
  while (!ready(Q, Qindex)) { }
 }
}
```

Figure 16. Listing 4: Second Stage After Loop Skewing

This example demonstrates how some transformations can expose opportunities for other transformations to run; in this case, software pipelining exposes opportunities for affine parallelization.

4.4 LLVM IR Processing and the Negotiator

Processing of the IR in LLVM is organized into passes, which are in turn categorized as analysis or transform passes. Analysis passes generally evaluate a unit of the IR to build an internal data structure (e.g., PDG or CDG) to be used by one or more transform passes in rearranging the IR. A transform may have no effect on some analysis results, or a transform pass may update an analysis, or the transform may invalidate the analysis – in which case the LLVM infrastructure will redo those analyses needed by subsequent passes.[2]

The position of the negotiator is flexible. Logically, the negotiator should execute before all of the parallelizing transformations, and after building the PDG. The PDG should be built after running the alias analysis passes, after Satisfiability Modulo Theories Alias Analysis (SMTAA) (Harris, et al. 2010) and after traditional loop transformations. SMTAA should execute after alias analysis and after loop transformations. Alias analysis should execute after loop transformations. Loop transformations should execute after all other traditional optimizations. Here is a plausible pass ordering:

- Traditional Optimizations
- Loop Optimizations
- Alias analysis

[2] See http://LLVM.org/docs/WritingAnLLVMPass.html

- SMTAA
- Build PDG
- Run Negotiator
- Do Parallelization

The exact ordering of the passes will change in the future as we develop new techniques. Note that remediators and critics are not passes. In LLVM, passes transform the LLVM IR. Fixers and complainers transform the PDG directly.

The optimization knowledge base is managed by a new analysis pass, and characterization information is unaffected by IR transforms. Profile information is invalidated by changes to the source code of the unit to which it refers. Performance model expressions will be derived (by a new analysis pass) for any code unit in the course of evaluating a candidate transform, and a new transform pass will use the performance expressions to choose among alternative transforms or to create runtime choices. A block diagram of how AESOP does IR processing is shown in Figure 17.

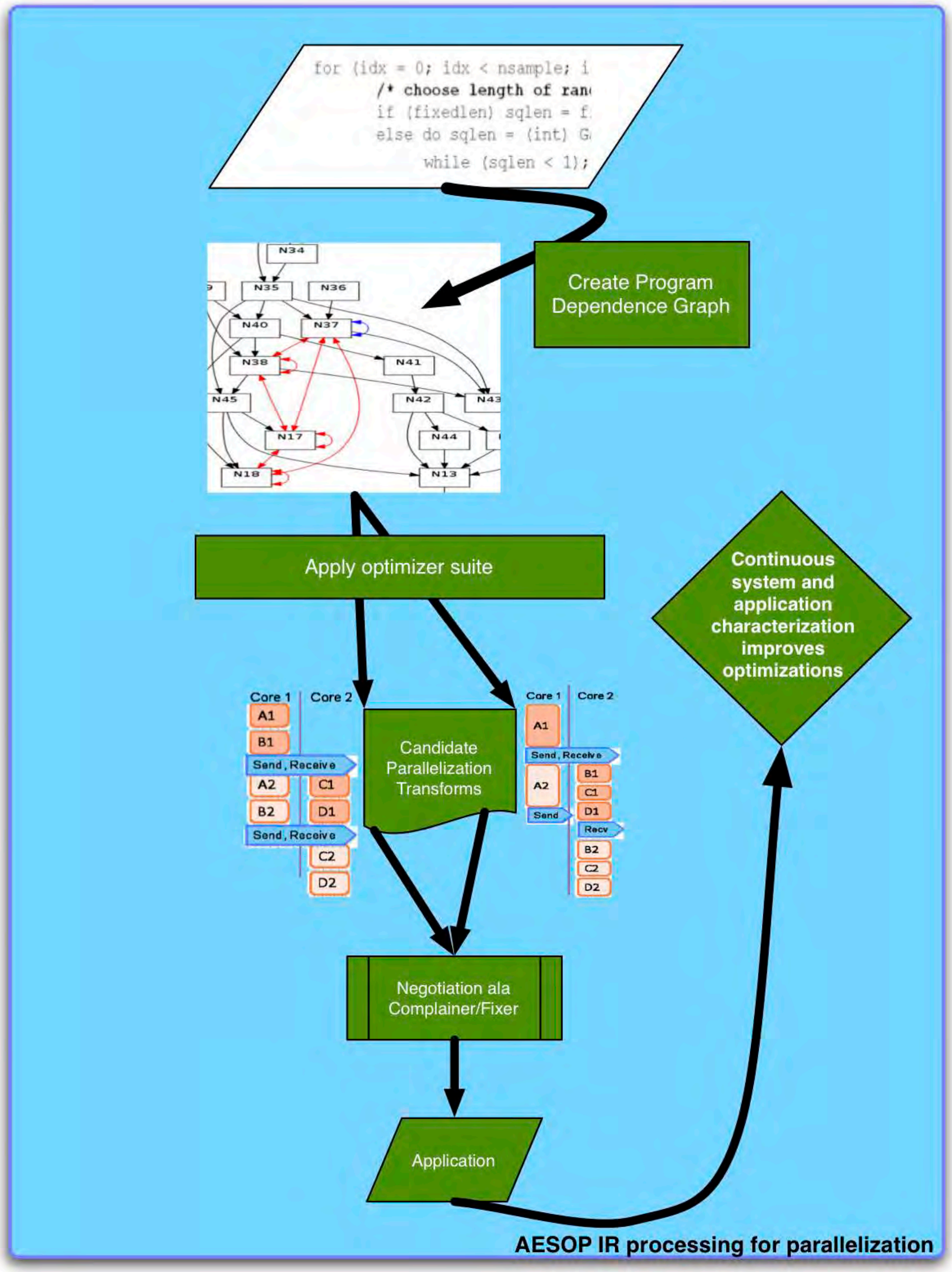

for (idx = 0; idx < nsample; i
/* choose length of ran
if (fixedlen) sqlen = f
else do sqlen = (int) G
while (sqlen < 1);
Create Program Dependence Graph
N34
N35
N36
N40
N37
N38
N41
N45
N42
N43
N17
N44
N18
N13
Apply optimizer suite
Core 1
Core 2
A1
B1
Send, Receive
A2
C1
B2
D1
C2
D2
Candidate Parallelization Transforms
Send
Recv
Continuous system and application characterization improves optimizations
Negotiation ala Complainer/Fixer
Application
AESOP IR processing for parallelization

Figure 17. AESOP's IR Processing

AESOP's IR Processing for Parallelization including the Negotiator Framework (ties to figure Figure 4)

4.5 Affine Parallelizer

The first of two parallelizers in AESOP is the affine parallelizer. One of the pressing problems in affine parallelization research has been determining the order of transformations to be applied. It is well known that different orders of transformations give different results. Different loops may require the transformations to be applied in a different order for maximum performance. Hence, fixing an order of transformations may constrain the system. On the other hand, searching for all combinations is an $O(n!)$ problem. Indeed it may be worse than $O(n!)$ since some transformations need to appear more than once in the best transformation order.

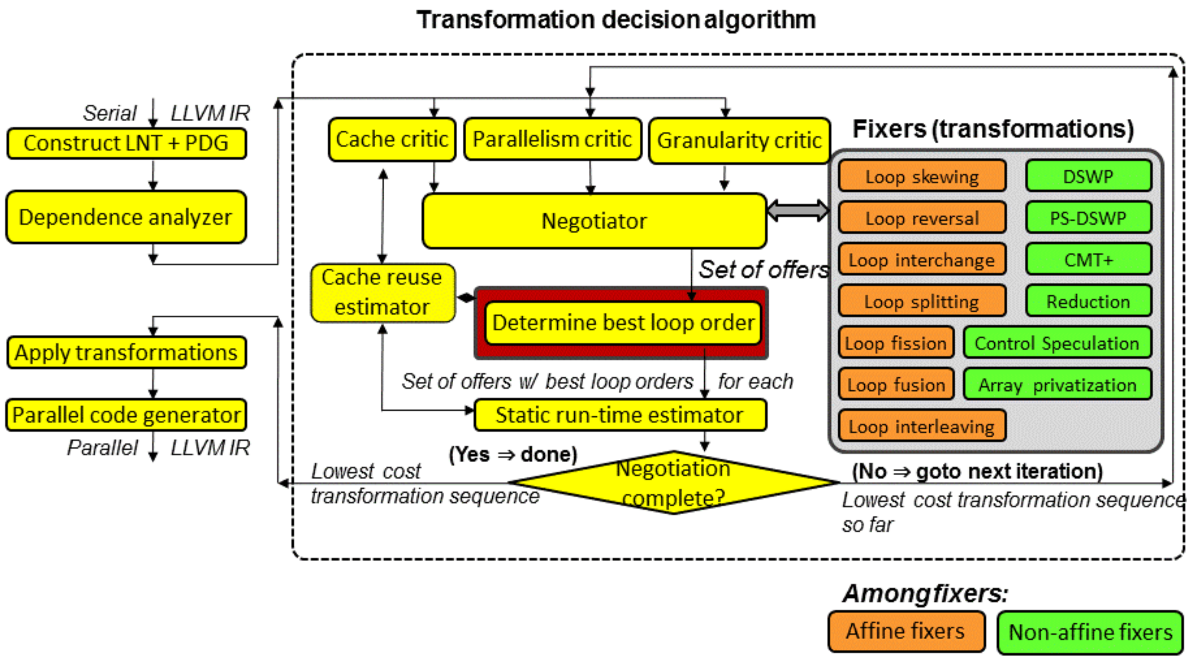


Figure 18. Decision Algorithm for Affine Parallelizer

Figure 18 shows the architecture of AESOP's affine parallelizer's transformation decision algorithm. On the top left, the serial program's IR is augmented with two higher-level hierarchical program representations that will help search the transformation space quickly – the LNT and the PDG. At the highest level is the Loop Nest Tree (LNT) (Yellareddy 2011), which is a new data structure we define. It shows the loop structure of the program, and has pointers to the well-known lower-level PDG representation which shows details such as the instructions and other control-flow at each loop nesting level. In turn, the PDG contains the pointers to the compiler IR of the program (the lowest-level representation). Next the dependence analyzer annotates the LNT with loop-carried distance vectors. Ordinary scalar data and control dependencies are already part of the PDG. The annotated program is then passed to the transformation decision algorithm.

The reason for this hierarchical representation of the program (LNT, PDG and IR) is to reduce the overhead of the transformation decision algorithm, which evaluates a large number of possible program transformations before choosing the best. Program transformations (especially loop-level affine transformations) can much more quickly change the LNT than the IR directly. For example, loop interchange can be done at low cost on the LNT, allowing the PDG and IR to remain unchanged. The PDG and IR are thereafter correct only in instructions but not loop structure. This is okay since the loop structure is looked up only in the LNT by the transformation search. After the final transformation order has been decided, the LNT, PDG and IR are all updated to correct any obsolete information.

The transformation decision algorithm has five major components: the critics, the negotiator, the remediators, the cache reuse model, and the cache and granularity scheduler. We briefly outline their role here; details are in the rest of this document. The critics are modules that identify problems that prevent good performance. The cache critic identifies cache performance problems, such as the inner loop not striding along cache lines, or consecutive loops accessing the same data, but not availing of reuse because their cache footprints exceed the cache size. The parallelism critic identifies loops which cannot be parallelized because of loop-carried dependencies. The granularity critic identifies situations where parallelism available is too fine-grained to be profitably exploited.

The problems identified by the critics are passed to the negotiator which looks for the correct series of remediators (code transformations) to fix those problems. Affine transformations include loop interchange, fusion, reversal, skewing, fission, and alignment. Non-affine transformations include DSWP, PS-DSWP, CMT+, and Control Speculation.

Examples of how remediators can fix performance problems are: the inner loop not striding along cache lines may be fixed by loop interchange; consecutive loops with shared data not taking advantage of cache reuse may be fixed by loop fusion; a cache footprint that exceeds the cache size may be fixed by loop fission; a loop-carried dependence can be broken by loop skewing, reduction, or loop fission; and the granularity of parallelism can be increased by loop interchange and fusion. Many other examples are possible.

The negotiator queries the remediators (transformations) to see if they can solve the problems identified by the critics. When multiple problems are identified for a loop, or when fixing one problem creates another, then multiple transformations may need to be applied in the correct sequence. In this case, the negotiator uses an intelligent goal-directed search to find a good order. It can come up with many possible reasonable solutions (called offers) which are passed to the next stage, the cache and granularity scheduler (CGS).

The CGS interchanges loop nest dimensions to find an order that is as close to the ideal order as possible (McKinley 1998). We augment that method to also find the best cache-optimal tiling strategy in the CGS. The static run-time estimator then estimates the run-time of each offer and chooses the one with the lowest run-time. Even this solution is not final. This entire process of critics, remediators, negotiator, CGS, and the cache reuse model may run multiple times in a loop, in the hope of finding a transformation order with lower estimated run-time, until a certain (low) threshold in number of iterations is reached.

Once the best order of transformations is found, they are applied to yield serial IR that is more parallelizable and cache-efficient. Next, parallelism is added to yield parallel IR. Finally the IR is translated to the output binary via the LLVM x86 backend, or to output C code via the LLVM C backend.

We made an explicit decision not to use the Polyhedral model in our affine compiler because we believe our model accomplishes much of what it achieves, and overcomes its main drawbacks. The Polyhedral model represents the affine accesses as polyhedra in an iteration space, and represents transformations as scheduling functions on this space. Although it is a powerful model to find transformation sequences, we have found that our model performs as well as the Polyhedral model for every example we could find in Polyhedral papers that shows its superiority over the traditional model. Moreover our model overcomes the two main drawbacks of the Polyhedral model: (i) unlike the Polyhedral model, our model has no worst-case exponential compile-time passes; and (ii) our model can parallelize mixed affine and non-affine loops in an elegant manner given the appropriate non-affine remediators; the Polyhedral model cannot handle affine loops with even small amounts of non-affine memory references.

4.5.1 Use of static system characteristics

Our affine parallelizer, when integrated with the graph-based parallelizer, uses the static performance estimator described earlier to compare the run-times of different possible parallelizations. In that sense it uses all the system characteristics that are used by the static performance estimator. In addition, the affine parallelizer will use at least the following system characteristics for its decision-making:

4.5.1.1 Number of parallel hardware contexts.

It is clear that the number of parallel hardware contexts will impact the affine parallelizer. In particular the affine parallelizer will generate a number of threads per parallel loop that will equal the number of parallel hardware contexts. This is supported by our experiments: when the number of threads exceeds the number of available parallel hardware contexts, performance tends to degrade rapidly, presumably because of context switching, and because affine loops cannot complete until all of the threads are completed, leading to the last thread determining run-time.

As described in the system characterization chapter, when the hardware contexts share memory or floating point resources, the number of efficiently usable floating point or memory contexts may be less than the number of hardware contexts available to threads without floating point and memory demands. To avoid excessive contention for such resources, when the affine loop in question is floating point or memory dominated (as determined by heuristics we shall explore), then we will use the lower number of hardware contexts available under that resource constraint for parallelization.

4.5.1.2 Choice of barriers and broadcasts.

As in most earlier affine parallelizers, we insert a broadcast at the start of each parallel loop, and a barrier at the end of each parallel loop, to ensure correctness in the face of dependencies between the loop and code outside the loop. This necessary synchronization introduces run-time overhead, which we aim to minimize using the best choice of multiple barrier and broadcast implementations available on the system for which we are compiling. We have implemented a variety of barriers, including centralized, static tree, butterfly and Pthreads barriers for AESOP. Our experiments show that there is no globally "best" barrier available; different barriers perform best depending on the machine in question and the number of threads participating in the barrier. Our system characterization collects the information of what barrier is best for a given target machine for different number of threads. This information is then used by our affine parallelizer to choose the appropriate barrier.

4.5.1.3 Cache characteristics.

Cache characteristics, such as the cache capacity, line size and associativity for each level of the cache, as well as memory layout, are used by the affine parallelizer to optimize for run-time. Specifically the cache size is used to see if the estimated memory footprint per thread of a certain transformed version of an affine loop will fit in the cache for that context. The memory footprint is not the same for all parallelized versions of the loop; for example, transformations like cache-aware multi-dimensional blocking can result in a lower memory footprint than single-dimensional blocking in matrix multiply. Our parallelizer will prefer transformed versions that minimize cache footprint (and hence maximize cache reuse). We have developed a memory footprint estimation module for this purpose. Multiple levels of cache can be accounted for by preferring partitions whose footprint fits in the highest level of cache possible. The cache line size is used to ensure that blocking is done in a way that eliminates or minimizes false sharing of cache lines between parallel contexts that use different primary caches. This is achieved by using block sizes that are multiples of the cache line size. The memory layout of the arrays (row-major or column-major), which is a compiler and language characteristic, used to prefer partitions where the inner loops stride along a cache line, which maximizes spatial reuse. Finally we are investigating methods to see whether associativity can be useful in affine parallelization.

Cache transformations are valuable even when parallelization is not possible, for example when the target has only one available hardware context, or when the loop cannot be parallelized. The transformations above to optimize for cache size and memory layout are naturally applicable even in such cases, and will be applied.

4.5.2 Affine transformations implemented

It is well known that a variety of loop transformations (M. E. Wolf 1991) (Bacon 1994) for affine loops can improve their performance. However the challenge has been to find an effective decision algorithm for which transformations to apply and in what order. Existing decision algorithms in traditional models (M. E. Wolf 1991) (McKinley 1998) are limited in several ways, in that they consider only a few transformations, use only a few system characteristics, and are based on heuristics. For example, McKinley's algorithm (McKinley 1998) – a leading decision algorithm in the literature – only considers blocking and loop interchange transformations ignoring others, and uses only memory layout for spatial locality among cache system characteristics.

We have designed a unified decision algorithm in AESOP that will simultaneously consider all the transformations listed in this section, and use all the system characteristics listed in the previous section. This unified decision algorithm is based on the critic-fixer model developed at Princeton University, augmented by specialized domain knowledge from the affine space. The decision algorithm will be presented in the next section.

This section lists the affine transformations that we will implement in AESOP, and intuitively describes the value of each transformation, and under what system characteristics it might be useful. The next section will describe more precisely how each transformation is integrated in our decision algorithm. Each of the transformations below (Bacon 1994) (Allen and Kennedy 2002) (Eigenmann 1991) has legality conditions that are well known, and hence not discussed.

4.5.2.1 Loop Interchange

The well-known loop interchange transformation changes the order of loop dimensions in multi-dimensional loop nests. Loop interchange, like some other transformations, may be beneficial in two ways. First, it may improve run-time by more effectively using system characteristics. For example, loop interchange can increase the granularity of parallelism by moving inner parallel loop dimensions outwards, reduce the memory footprint of the parallel loop per thread, or move a loop dimension that strides along the cache line to the innermost dimension. Second, loop interchange can be used to eliminate or overcome loop-carried dependencies. For example, after interchange some loop-carried dependencies may no longer need to be independently enforced since they may be automatically enforced when another loop dimension is serialized (Allen and Kennedy 2002). AESOP will measure these benefits in the light of available system characteristics to decide when and where to apply these benefits.

4.5.2.2 Loop Fission

Loop fission breaks a loop into two more loops by splitting its body among successive loops. Loop fission can be used to reduce the memory footprint of a loop since fewer memory references will be in each loop after fission. A reduced memory footprint may fit better in cache, promoting reuse, but loop fission comes at the cost of increased looping overhead. Effectively trading off these competing benefits and costs will be accomplished by our critic-fixer model, which will measure benefits and costs using a cost model for how long different versions of a loop take to execute.

Loop fission may also be useful to break loop-carried dependencies. For example, if a loop iteration writes to A[i] but reads from A[i-2], then this loop-carried true dependence may be broken if the two array references are placed in different loops after fission. Our decision algorithm will consider both these different uses of loop fission.

4.5.2.3 Loop Fusion

Loop fusion combines two or more loops, usually adjacent loops, into a single merged loop with a combined loop body. Loop fusion may be beneficial because it reduces loop overheads, and may increase cache reuse because of accesses to the same memory locations in corresponding iterations of to-be-fused loops. It may also help reduce memory bandwidth demand since the fused loops may have fewer memory references per unit time, which is valuable if the memory bandwidth limit of the target machine is saturated by the loop. On the other hand, loop fusion may reduce performance since it may increase the memory footprint of a single parallel thread after fusion. This may be harmful if the memory footprint of the fused loops exceeds the cache size of the target processor. Making these tradeoffs requires system characteristics of the target. Our decision algorithm considers all these factors when deciding when to apply fusion.

4.5.2.4 Reductions

The well-known reduction transformation is possible on a loop dimension when a statement of the form v = v □ expr is present in a loop, operator □ is associative and commutative, and v is loop-invariant in that loop dimension. This dependence can be broken by creating a copy of v on each thread, leading to a parallelizable loop, followed by a new loop to accumulate all the copies of v into the original v using operator □. Reductions are useful to increase the degree of parallelism available in all kinds of loops including non-affine loops, but increase in effectiveness when the rest of the loop can be proven parallel using affine methods. Reduction is considered together with other affine transformations to improve affine loops in our critic-remediator model.

4.5.2.5 Array Privatization

Array privatization (Eigenmann 1991) refers to the process of cloning a private copy of an array for each parallel thread corresponding to a single array in the serial program. It is useful especially in two cases. First, it is useful when all the updates of the array are used in the same loop iteration or in the iteration space local to the processor. Second, it is useful with array reductions, i.e., a reduction operation on elements of the array; in this case, privatizing the array for every thread can break the dependence. However, the profitability due to parallelization as a result of array privatization may be over-shadowed by the time taken for memory initialization. These two factors will be taken into account while calculating the profitability due to array privatization.

4.5.2.6 Loop Skewing

Loop skewing is when the iteration space is partitioned along diagonal boundaries. It is useful when loop-dependencies prevent parallelization along any one dimension; for example, with a distance vector of (1,1), parallelization is not possible along either the outer or inner loop, but is possible along 45° diagonals. Loop skewing is used in AESOP as a fixer to break such dependencies, with the integrated decision algorithm deciding when it should be applied.

4.5.2.7 Loop Reversal

Loop reversal is when a loop dimension is transformed such that its order of computing iterations is the reverse order of the original loop. Loop reversal improves the applicability of other transforms such as loop interchange and loop skewing. Loop reversal is thus used in AESOP as a remediator which improves the applicability and/or profitability of the loop in isolation or in conjunction with other remediators.

4.5.2.8 Loop splitting

Loop splitting is a transformation in which the complete set of iterations in a loop are split into two or more parts and run serially. This could be a very useful transformation to break dependencies. We have presented examples in section 6.1 that benefit from loop splitting.

4.5.2.9 Loop interleaving

Loop interleaving is a loop transform that splits the iterations of the loop not into chunks but groups interleaving iterations together. This can also be very useful in breaking certain dependencies as shown in section 6.1.

4.5.2.10 Blocking / Strip mining

A loop dimension is said to be strip-mined when it is split into two nested loops, with the inner loop computing a block (sub-range) of iterations, and the outer loops iterating across blocks. The outer loop may be eliminated if that loop is parallelized, since the outer loop's induction variable may be replaced by the current parallel context number, or some value computed from it. In that sense strip mining is the essential basic step for converting any loop to a parallel loop.

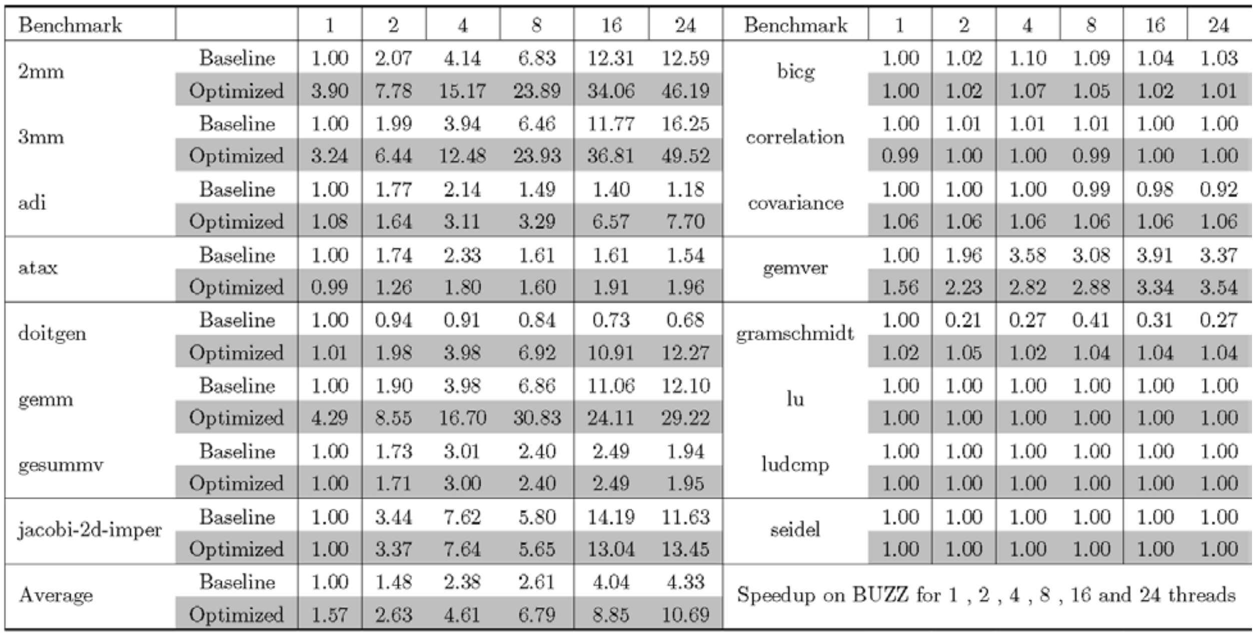
Since more than one original loop dimension may be strip-mined (blocked), strip-mining allows more than one parallel loop dimension, leading to more choices of how the iteration space may be partitioned. For example, in addition to row- and column-length blocks, blocks that simultaneously partition both rows and columns become possible. Strip-mining is valuable since some of these new blocking possibilities may have lower memory footprint, and hence better cache performance than row- and column-length blocks. Strip-mining is also valuable since after strip-mining, other loop transformations such as interchange have a larger space of possibilities to consider, some of which may lead to better run-time. Our model will account for both these benefits of strip-mining.

4.5.3 Results

Because the AESOP project was cancelled before completion, the results described here are preliminary. We use benchmarks from the Polybench benchmark suite to evaluate how well our affine parallelization strategy is working, and in particular, the affine decision algorithm which finds good transformation sequences. The Polybench benchmarks represent heavily used kernels in scientific and multi-media workloads and are suited for affine analysis.

Table 1 shows results of parallelism and cache-optimization on these benchmarks. As can be seen, several of the benchmarks give substantial improvement even for the single thread case due to improvement in cache locality from using loop interchange and fusion, which are the main transformations implemented so far that yield benefit. The results demonstrate that our prototype compiler has been correctly built using our program representation, and can find runtime-improving transformations.

Table 1. Speedups With and Without Transformations for Polybench Benchmarks



Benchmark		1	2	4	8	16	24	Benchmark	1	2	4	8	16	24
2mm	Baseline	1.00	2.07	4.14	6.83	12.31	12.59	bicg	1.00	1.02	1.10	1.09	1.04	1.03
	Optimized	3.90	7.78	15.17	23.89	34.06	46.19		1.00	1.02	1.07	1.05	1.02	1.01
3mm	Baseline	1.00	1.99	3.94	6.46	11.77	16.25	correlation	1.00	1.01	1.01	1.01	1.00	1.00
	Optimized	3.24	6.44	12.48	23.93	36.81	49.52		0.99	1.00	1.00	0.99	1.00	1.00
adi	Baseline	1.00	1.77	2.14	1.49	1.40	1.18	covariance	1.00	1.00	1.00	0.99	0.98	0.92
	Optimized	1.08	1.64	3.11	3.29	6.57	7.70		1.06	1.06	1.06	1.06	1.06	1.06
atax	Baseline	1.00	1.74	2.33	1.61	1.61	1.54	gemver	1.00	1.96	3.58	3.08	3.91	3.37
	Optimized	0.99	1.26	1.80	1.60	1.91	1.96		1.56	2.23	2.82	2.88	3.34	3.54
doitgen	Baseline	1.00	0.94	0.91	0.84	0.73	0.68	gramschmidt	1.00	0.21	0.27	0.41	0.31	0.27
	Optimized	1.01	1.98	3.98	6.92	10.91	12.27		1.02	1.05	1.02	1.04	1.04	1.04
gemm	Baseline	1.00	1.90	3.98	6.86	11.06	12.10	lu	1.00	1.00	1.00	1.00	1.00	1.00
	Optimized	4.29	8.55	16.70	30.83	24.11	29.22		1.00	1.00	1.00	1.00	1.00	1.00
gesummv	Baseline	1.00	1.73	3.01	2.40	2.49	1.94	ludcmp	1.00	1.00	1.00	1.00	1.00	1.00
	Optimized	1.00	1.71	3.00	2.40	2.49	1.95		1.00	1.00	1.00	1.00	1.00	1.00
jacobi-2d-imper	Baseline	1.00	3.44	7.62	5.80	14.19	11.63	seidel	1.00	1.00	1.00	1.00	1.00	1.00
	Optimized	1.00	3.37	7.64	5.65	13.04	13.45		1.00	1.00	1.00	1.00	1.00	1.00
Average	Baseline	1.00	1.48	2.38	2.61	4.04	4.33	Speedup on BUZZ for 1 , 2 , 4 , 8 , 16 and 24 threads						
	Optimized	1.57	2.63	4.61	6.79	8.85	10.69							

To show the possibility of combining affine and non-affine transformations, consider the example program in Table 2 which benefits from both DSWP and loop interchange. The results in Table 2 show that the speedup of the program is better when using affine and non-affine transformations – interchange and DSWP – in combination (5.8X) as compared to using either interchange only (2.8X) or non-affine only (1.1X). Whereas the DSWP partitions the loop such that the inner loop is on one thread and the "computation()" on the other thus reducing the runtime, interchange modifies the cache access pattern of one of the pipeline stages to improve cache reuse, and thus reduce runtime. Thus in combination, the transformations yield a better result.

Table 2. Example Using Affine and Non-Affine Transformations

```
for (x=0; x<Tot; ++x) {
    acc=0;
    computation();
    for(i=0; i<M; i++) {
        for (j=0; j<N; j++) {
            acc = acc + arr[j][i];
        }
    }
 Arr[x] = y + acc;
}
```

Transformations (#Thr)	#Thr	Speedup
Base(1)	1	1
DSWP(2)	2	1.1
Affine(1)	1	2.8
DSWP (2) + Affine(1)	2	3.4
DSWP(2) + Affine(2)	3	5.8
DSWP(2) : DSWP transformation with 2 threads Affine(2) : Affine Transformations with 2 threads		

4.6 Software Pipelining Parallelizer

The second of the two parallelizers implemented in AESOP to date is the software pipelining parallelizer. Figure 19 shows its architecture. The input to the system is the sequential application in the form of LLVM IR, which is generated from C by the LLVM frontend (not shown). First, the PDG is built from the IR in order to represent all of the control and data dependencies in the application, as described in section 0. If the program contains developer annotations such as the commutative annotation (to be described in section 4.6.3), the dependencies in the PDG will be updated to reflect these annotations.

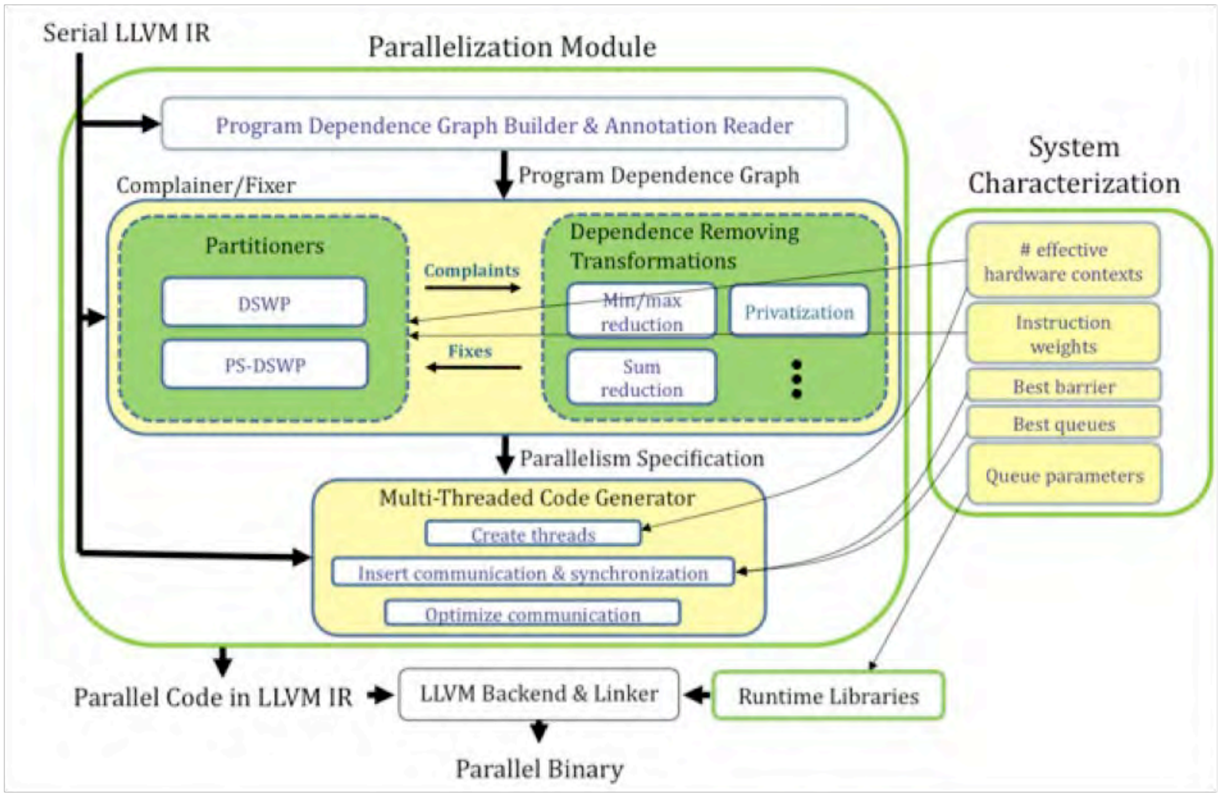


Figure 19. Software Pipelining Parallelizer

Block diagram of software pipelining parallelizer.

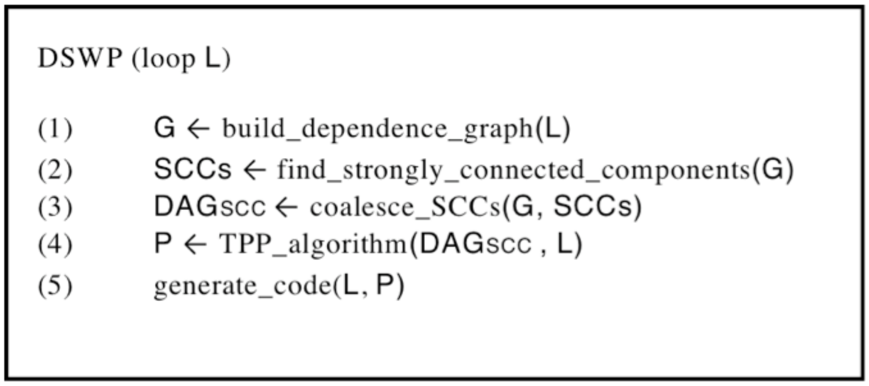
Next, the complainer/fixer module manages the partitioners, which perform the actual software pipelining, and the dependence removers, which perform code transformations to facilitate the partitioners. In this phase, transformations such as min/max reduction, sum reduction, and array privatization are performed in order to allow the partitioners (DSWP and PS-DSWP) to best parallelize the code.

After the complainer/fixer module determines the best transformation and parallelization strategy for the application, the Multi-Threaded Code Generator (MTCG) creates the parallel code according to this strategy. Individual functions are created, each containing code that will be executed by one of the threads in the parallel execution. Communication and synchronization code is inserted between these parallel threads, and between the parallel region and the preceding and succeeding sequential regions. Finally, the output of the parallelization module is linked with runtime libraries, including the queue library, producing an executable binary.

System characterization parameters are used by several of the modules in the parallelizer. The number of effective hardware contexts is used by the partitioners in order to determine how best to partition the code for the given number of contexts. Instruction weights help the partitioners to achieve a balanced software pipeline. Information about barriers and queues helps MTCG to decide what implementations of these items should be used on a given system. Queue parameters are important in tuning queue performance for a given system. Further details about the use of system characterization parameters in the parallelization system will be given in section 4.6.9.

4.6.1 Decoupled Software Pipelining (DSWP)

The first step in the DSWP algorithm (listed in Figure 20) is to build the dependence graph G for loop L, as described in section 4.2. The second step in the algorithm is to ensure an acyclic partitioning by finding the strongly connected components (SCCs). The SCCs correspond to instructions collectively participating in a dependence cycle, the loop recurrences. As such, DSWP requires all instructions in the same SCC to remain in the same thread. Step (3) coalesces each SCC in G to a single node, obtaining the DAGSCC. Using the concepts above, we now define a valid partitioning of the DAGSCC.



```
DSWP (loop L)

(1)     G ← build_dependence_graph(L)
(2)     SCCs ← find_strongly_connected_components(G)
(3)     DAGscc ← coalesce_SCCs(G, SCCs)
(4)     P ← TPP_algorithm(DAGscc , L)
(5)     generate_code(L, P)
```

Figure 20. The DSWP Algorithm

PROPERTY 1: VALID DSWP ASSIGNMENT

A valid partitioning P of the DAGSCC is a sequence P_1, P_2,..., P_n of sets of DAGSCC's vertices (i.e. P_is are sets of SCCs) satisfying the following conditions:

1. *$1 \leq n \leq t$, where t is the number of threads that the target processor can execute simultaneously.*
2. *Each vertex in DAGSCC belongs to exactly one partition in P.*
3. *For each arc $(u \rightarrow v)$ in DAGSCC, with $u \in P_i$ and $v \in P_j$, we have $i \leq j$.*

Figure 21. Property 1: Valid DSWP Assignment

A valid assignment guarantees that all members of partition $P_i \in$ P can be assigned to a thread loop $L_i$and that this loop L_i can be executed in its own context. Condition (3) in Property 1 guarantees that each arc in the dependence graph G either flows forward to a loop L_j where $j > i$ or is internal to its partition. In other words, this condition guarantees an ordering between the partitions that permits the resulting loops to form a pipeline.

The Thread-Partitioning Problem (TPP) is the problem of choosing a valid partitioning that minimizes the total execution time of the resulting code. The optimal partitioning of the DAGSCC that minimizes this cost is machine dependent, and can be demonstrated to be NP-complete through a reduction from the bin packing problem. In practice, we use a heuristic to maximize the load balance among the threads. This is a commonly used criterion in scheduling and parallelization problems and generally performs well here. As in a processor pipeline, the more balanced the DSWP stages are, the greater its efficiency. In other words, the thread pipeline is limited by the stage with the longest average latency.

Our heuristic computes the estimated cycles necessary to execute all the instructions in each SCC by considering the instruction latency and its execution profile weight. Function call latencies should include the average latency to execute the callee. The algorithm keeps a set of candidate nodes, whose predecessors have already been assigned to a partition, and proceeds by choosing the SCC node in this set with the largest estimated cycles. When the total estimated cycles assigned to the current partition (P_i) gets close to the overall estimated cycles divided by the desired number of threads, the algorithm finishes partition P_i and starts assigning SCC nodes to partition P_{i+1}. In order to minimize the cost of necessary flows between the threads, the heuristic breaks ties by choosing a candidate SCC that will reduce the number of outgoing dependencies from the current partition.

Once each SCC has been assigned to a thread, the final step is to generate the parallel code. This is done by the Multi-Threaded Code Generator (MTCG), described in section 4.6.7.

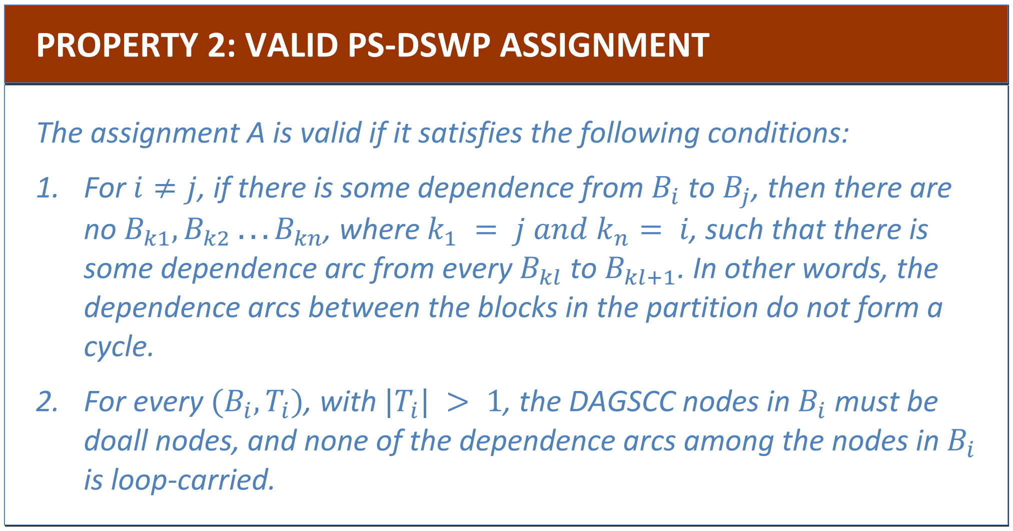
4.6.2 Parallel Stage-Decoupled Software Pipelining (PS-DSWP)

DSWP is limited in its ability to extract DOALL-like, data-level parallelism. The Parallel-Stage extension to DSWP removes this limitation by allowing pipeline stages with no inter-iteration dependencies to be replicated an arbitrary number of times. This section discusses the alterations to the DSWP algorithm necessary to achieve this end.

Let $D = \{d_1, d_2, \dots, d_k\}$ be the set of nodes in the DAGSCC. Let the number of target threads be denoted by n, and the set of threads be $T = \{t_1, t_2, \dots, t_n\}$. The thread partitioning algorithm takes D and T as input and produces the following as output:

- A partition $P = \{B_1, B_2, \dots, B_l\}$ of the nodes in the DAGSCC. Each element of P corresponds to one of the l stages of the pipeline.
- An assignment $A = \{(B_1, T_1), (B_2, T_2), \dots, (B_l, T_l)\}$, which maps the blocks in the partition P to subsets T_i of T. The T_is in fact partition the thread set T.

To be valid, an assignment A obtained as above must respect the following property:


PROPERTY 2: VALID PS-DSWP ASSIGNMENT

The assignment A is valid if it satisfies the following conditions:

1. *For $i \neq j$, if there is some dependence from B_i to B_j, then there are no $B_{k1}, B_{k2} \dots B_{kn}$, where $k_1 = j$ and $k_n = i$, such that there is some dependence arc from every B_{kl} to B_{kl+1}. In other words, the dependence arcs between the blocks in the partition do not form a cycle.*
2. *For every (B_i, T_i), with $|T_i| > 1$, the DAGSCC nodes in B_i must be doall nodes, and none of the dependence arcs among the nodes in B_i is loop-carried.*

Figure 22. Property 2: Valid PS-DSWP Assignment

The first condition ensures that the blocks B_i can be mapped to threads that form a pipeline. The second condition ensures that only doall nodes are present in a parallel stage and that there are no loop-carried dependencies within a parallel stage.

The goal of thread partitioning is to find an assignment that minimizes the execution time. Finding the optimal solution to this problem is NP-hard, even assuming that the execution times of operations are known a priori. For this reason, a heuristic solution is used to solve the partitioning problem in this work, focusing on loops that have a good amount of iteration-level parallelism. Hence, in this section, we simplify the partitioning problem by allowing only one of the pipeline stages to be assigned to more than one thread. In other words, the partitioning algorithm allows only one (B_i, T_i) with $|T_i| > 1$ and, for all other (B_j, T_j) pairs, $|T_j| = 1$. The remainder of this section describes the heuristic used to form pipelines with a single parallel stage. The algorithm will be generalized to form pipelines with multiple parallel stages if and when it is determined that such pipelines are necessary to achieve desired performance levels for applications of interest.

```
Input: $DAG_{SCC}, T$
Assign each node $i$ to its own block $B_i$
Initial partition $P = \{B_i | i \quad DAG_{SCC}\}$
Classify each $B_i$ as either doall or sequential
$D$ = merge_doall_blocks($P$)
$MAXD$ = max_profile_weight_block($D$)
Reassign members of the set $D-MAXD$ as sequential
$SEQ$ = merge_sequential_blocks($P$)
$d = |T|-|SEQ|$
$i=1$
$A = \{\}$
for $n \quad DAG_{SCC}$ in topological order do
    if block $B$ containing n is sequential then
        Add $(B,\{t_i\})$ to $A$
        $i = i+1$
    else
        Add $(B,\{t_i,t_{i+1},\ldots t_{i+d-1}\})$ to $A$
        $i = i+d$
    end if
end for
```

Figure 23. PS-DSWP Thread Partitioning Algorithm

Figure 23 shows the thread-partitioning algorithm. The algorithm starts by assigning each node in the DAGSCC to its own partition block. If a node is labeled **doall**, the corresponding block is also labeled as **doall**. The partitioning algorithm then greedily merges **doall** blocks. Two **doall** blocks B_1 and B_2can be merged if the following conditions are satisfied:

1. There is no block B_3 such that a node in B_3 is reachable from a node in B_1 and a node in B_2 is reachable from a node in B_3 in the DAGSCC. If such a block B_3 exists, then the dependencies between the blocks will form a cycle among the blocks after merging, resulting in a dependence cycle among the threads.

2. None of the dependence arcs connecting the PDG nodes in B_1 and B_2 is a loop-carried dependence arc. This is necessary to satisfy condition 2 of Property 2.

This process is continued until no more merging is possible. After merging, only the **doall** block with the maximum profile weight is retained as a **doall** block, while all remaining blocks are re-labeled as *sequential*. The sequential blocks are then merged greedily till no more merging is possible without violating the conditions for obtaining a valid assignment.

Each of the sequential blocks is assigned a single thread. All the remaining threads are assigned to the sole `doall` block.

As mentioned previously, this results in an assignment with one parallel stage, namely the `doall` block with the maximum profile weight. However, for some applications, it is beneficial to have more than one parallel stage. In particular, some of the blocks labeled sequential may have actually qualified as `doall`, but were not mergeable with the maximum weight `doall` block. In such cases, a pipeline with multiple parallel stages may be constructed. The allocation of threads to stages is then made with the goal of maximizing estimated performance by balancing the stages given their relative profile weights.

4.6.3 Enhancing the Applicability of Pipeline Parallelism

The partitioners, including DSWP and PS-DSWP, work on the program dependence graph, and must respect the dependence edges in that graph. The edges in the graph may restrict the ability of the partitioners to obtain high-performance parallel code. For example, large dependence cycles will prevent DSWP from obtaining balanced pipelines, and the lack of `doall` nodes will prevent PS-DSWP from obtaining large parallel stages. This section discusses techniques used to break dependence edges, enabling the partitioners to better parallelize the code.

4.6.4 Commutative Annotations

A challenge with automatic parallelization of sequential programs is that C forces the programmer to over-constrain the program. The programmer is forced to do this by providing only one valid result for the program even while others are equally valid. For example, consider a DOALL loop with a call to the C library function `malloc`. Since `malloc` has side effects, the compiler must respect the invocation order of `malloc` even though the programmer typically does not care about the specific addresses that are returned. This over-constrains the parallelization problem and leads to poor results.

This problem may be solved by using programmer annotations. Starting with a sequential C program, the programmer can add simple annotations, such as the commutative annotation, to certain regions of code to relax these over-constraints. When the compiler knows, for example, that calls to `malloc` are commutative, it is free to ignore the inter-iteration dependence edges at `malloc` callsites and can generate parallel code where the order of calls to `malloc` may differ from the original sequential order.

Listing 5 shows another example of the commutative annotation breaking a self-dependence edge. The code is a simplified version of the main loop in HMMER, a protein sequence analysis program, where calls to rand within the `RandomSequence` function create a self-dependence edge. In this example, a commutative annotation (marked by `#pragma` statement) tells the compiler multiple instances of `RandomSequence` can be arbitrarily interleaved in execution, and helps find an optimal pipeline structure to achieve best performance.

Listing 5
Simplified version of the main loop body of HMMER with commutative annotation

```
for (idx = 0; idx < nsample; idx++) {

  /* choose length of random sequence */
  sqlen = ...

  /* generate it */
#pragma CommutativeSet(SELF) /* “commutative” annotation
*/
  {
   /* RandomSequence calls rand function in it */
   seq = RandomSequence(Alphabet, randomseq,
Alphabet_size, sqlen);
  }

  /* rest of the loop body follows */
  ...
```

Figure 24. Listing 5: Simplified Version of the Main Loop Body of HMMER with Commutative Annotation

Support for this feature is added to the parallelizer in two places. First, the frontend is augmented to recognize the annotations and pass them from the C code into the LLVM IR. Second, the parallelizer itself is enhanced to understand the annotations in the LLVM IR and modify the edges in the program dependence graph accordingly. Once the information is reflected in the program dependence graph, the other modules in the parallelizer, including the partitioners and the code generator, will parallelize the code properly.

4.6.5 Privatization

Another source of inter-iteration dependencies that constrain parallelism is reuse of global (or out-of-loop scope) memory variables by different iterations of a loop, when the data is only “live” within an iteration. In such a case, anti (write-after-read) and output (write-after-write) dependencies may limit the amount of parallelism available across iterations of the loop. By privatizing such variables, that is, giving each iteration its own version of the variable, these dependencies may be broken, exposing more parallelism.

The compiler is able to determine if a variable V can be privatized by leveraging standard analyses to verify that, for every load from V in the loop, there is a store to V earlier in the same iteration of the loop. Once this is determined, the transformation is made such that each iteration uses its own copy of V. For scalar variables, this transformation can be thought of as a loop-aware register promotion of V.

When the variable to be privatized is an aggregate such as an array, a more elaborate scheme is used in order to reduce the amount of data potentially passed between stages in the software pipeline. Rather than pass the contents of the aggregate from stage to stage, it is sufficient to store the contents of the aggregate to a private location in memory and then pass the pointer between stages. To support this functionality, the compiler converts accesses to the original aggregate into accesses to a memory region representing an iteration-private version of the aggregate. The memory region can be thought of logically as being obtained from the heap in every iteration, but it is actually implemented by a runtime library as a rotating buffer in order to reduce performance overhead.

4.6.6 Reductions

Often, a stage in the software pipeline is not a parallel stage due to the presence of code that computes a minimum, maximum, or sum of values across iterations. Such computations induce an inter-iteration dependence and restrict parallelism. The compiler is able to recognize such computations and break the dependencies by inserting reductions. For sums, the sum reduction transformation (also known as accumulator expansion) changes the code so that each thread computes its own local sum; after the loop finishes executing, the local sums are added together to give the final global sum. For min/max reduction, the transformation is similar, with care taken so that the final min/max computation correctly handles cases where the local min/max values from different threads are the same.

4.6.7 Multi-Threaded Code Generator (MTCG)

This section describes a MTCG algorithm the AESOP team developed and implemented. This algorithm produces efficient and correct multi-threaded code for an arbitrary partition of the PDG nodes into threads. A key property of MTCG is that it does not replicate the control flow entirely in all threads. This property is key to enabling truly thread-level parallelism. To improve the performance of generated multi-threaded codes, inter-thread communication optimizations are applied to reduce the number of inter-thread communications.

4.6.7.1 MTCG Algorithm

The pseudo-code of the MTCG algorithm is illustrated in Figure 25. This algorithm takes as input a PDG for the code region being scheduled and a given partition (P) of its nodes (instructions) into threads. Furthermore, for simplicity, we assume here that a CFG representation of the original code region is also given. As output, this algorithm produces, for each of the resulting threads, a new CFG containing its corresponding instructions and the necessary communication instructions.

```
Require: $CFG, PDG, \mathcal{P} = \{T_1, \ldots, T_n\}$
1:  for each $T_i \in \mathcal{P}$ do                                             // 1. create CFG_i's basic blocks
2:      $V_{CFG_i} \leftarrow create\_relevant\_basic\_blocks(CFG, PDG, T_i)$
3:  end for
4:  for each instruction $I \in V_{PDG}$ do                                       // 2. put instruction in its thread's CFG
5:      let $i \mid I \in T_i$
6:      $add\_after(CFG_i, point_i(I), I)$
7:  end for
8:  $COMM \leftarrow \emptyset$
9:  for each arc $(I \rightarrow J) \in PDG$ [incl. transitive control dependences] do      // 3. insert communication
10:     let $T_i, T_j$ be such that $I \in T_i$ and $J \in T_j$
11:     if $(T_i = T_j) \vee ((I, T_j) \in COMM)$ then
12:         continue
13:     end if
14:     $COMM \leftarrow COMM \cup \{(I, T_j)\}$
15:     $q \leftarrow get\_free\_queue()$
16:     if $dependence\_type(I \rightarrow J) = Register$ then                    // register dependences
17:         $r_k \leftarrow dependent\_register(I \rightarrow J)$
18:         $add\_after(CFG_i, I,$ "produce $[q] = r_k$")
19:         $add\_after(CFG_j, I,$ "consume $r_k = [q]$")
20:     else if $dependence\_type(I \rightarrow J) = Memory$ then                 // memory dependences
21:         $add\_after(CFG_i, I,$ "produce.rel $[q]$")
22:         $add\_after(CFG_j, I,$ "consume.acq $[q]$")
23:     else                                                                       // control dependences
24:         $r_k \leftarrow register\_operand(I)$
25:         $add\_before(CFG_i, I,$ "produce $[q] = r_k$")
26:         $add\_before(CFG_j, I,$ "consume $r_k = [q]$")
27:         $add\_after(CFG_j, I, duplicate(I))$
28:     end if
29: end for
30: for each $T_i \in \mathcal{P}$ do
31:     for each live-in register $r$ in $T_i$ do                                  // initial communication
32:         $q \leftarrow get\_free\_queue()$
33:         $add\_last(START,$ "produce $[q] = r$")
34:         $add\_last(START_i,$ "consume $r = [q]$")
35:     end for
36:     for each live-out register $r$ defined in $T_i$ reaching $END$ do          // final communication
37:         $q \leftarrow get\_free\_queue()$
38:         $add\_last(END_i,$ "produce $[q] = r$")
39:         $add\_last(END,$ "consume $r = [q]$")
40:     end for
41:     $q \leftarrow get\_free\_queue()$
42:     $add\_last(END_i,$ "produce.rel $[q]$")
43:     $add\_last(END,$ "consume.acq $[q]$")
44: end for
45: for each $T_i \in \mathcal{P}$ do                                             // 4. create CFG_i's arcs
46:     for each branch instruction $I \in T_i$ do
47:         $redirect\_target(I, closest\_relevant\_postdom_i(target(I)))$
48:     end for
49:     for each $B \in V_{CFG_i}$ do
50:         CRS $\leftarrow closest\_relevant\_postdom_i(succ(orig\_bb(B)))$
51:         $add\_last(B,$ "jump CRS")
52:     end for
53: end for
```

Figure 25. Pseudocode for the MTCG Algorithm

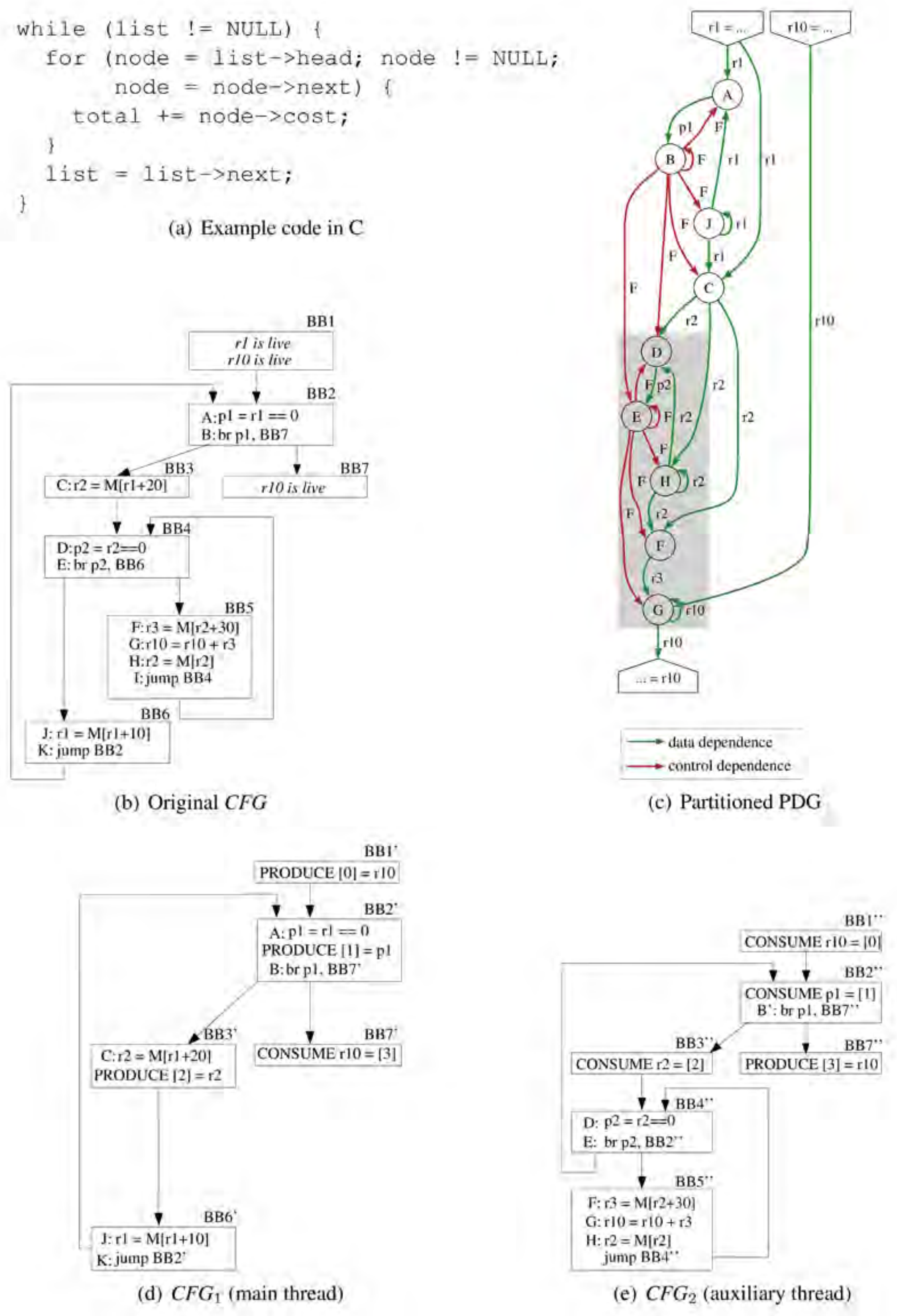
The thread model assumed here is that the program executes sequentially until it reaches a region that has been scheduled on multiple threads. Upon reaching such parallelized region, auxiliary threads are spawned, and each of them will execute one of the CFGs produced by MTCG. In this model, auxiliary threads do not spawn more threads. The main thread (which may also execute one of the new CFGs) then waits for completion of all the auxiliary threads. Once all auxiliary threads terminate, sequential execution resumes. This thread model is known as fork-join. In practice, the cost of spawning and terminating threads can actually be reduced by creating the threads only once, at the start of the program. The threads then execute a loop that continuously waits on the address of the code to be executed and invokes the corresponding code, until they are signaled to terminate.

Before going into the details in Figure 25, let us introduce the notation used. T_i denotes a block (thread) in P, and CFG_i denotes its corresponding control-flow graph. For a given instruction I, $bb(I)$ is the basic block containing I in the (original) CFG, and $point_j(I)$ is the point in CFG_j corresponding to the location of I in CFG.

In essence, the MTCG algorithm has four main steps. As we describe these steps, we illustrate them on the example in Figure 26(a) contains the source code in C, and Figure 26(b) contains the corresponding code in a low-level IR. We illustrate how MTCG generates code for the partitioned PDG in Figure 26(c), where the nodes in the shaded rectangle are assigned to one thread, and the remaining instructions to another. In the PDG, data dependence arcs are annotated with the corresponding register holding the value, while control dependence arcs are labeled with the corresponding branch condition. In this example, there are no memory dependencies. Special nodes are included in the top (bottom) of the graph to represent live-in (live-out) registers. The resulting code for the two threads is illustrated in Figure 26(d) and Figure 26(e). In this example, the main thread also executes the code for one of the generated threads (CFG1).

4.6.7.2 STEP 1: CREATING NEW CFGS' BASIC BLOCKS

MTCG creates a new CFG (CFG_i) for each thread T_i. Each CFG_icontains a basic block for each relevant basic block to $T_i$in the original CFG. The notion of relevant basic block is defined below. In its last step, described in Step 4, MTCG adds the arcs connecting the basic blocks in each CFG_i. These arcs are inserted in a way that guarantees the equivalence between the condition of execution of each new basic block and its corresponding block in the original CFG.



```
while (list != NULL) {
  for (node = list->head; node != NULL;
       node = node->next) {
    total += node->cost;
  }
  list = list->next;
}
```

(a) Example code in C

(b) Original *CFG* (c) Partitioned PDG

(d) CFG_1 (main thread) (e) CFG_2 (auxiliary thread)

Figure 26. Example of the MTCG Algorithm

Example of the MTCG algorithm including (a) C source code; (b) original low-level code; (c) partitioned PDG; and (d)-(e) resulting-threaded code.

DEFINITION 1: RELEVANT BASIC BLOCK

A basic block B is relevant to a thread T_i if B contains either:

1. *an instruction scheduled to T_i; or*
2. *an instruction on which any of T_i's instructions depends (i.e. a source of a dependence with an instruction in T_i as the target); or*
3. *a branch instruction that controls a relevant basic block to T_i.*

Figure 27. Definition 1: Relevant Basic Block

The reason for including basic blocks containing instructions in T_i is obvious, as they will hold these instructions in the generated code. The reason for adding the basic blocks containing instructions on which T_i's instructions depend is related to where communication instructions are inserted by MTCG, as described shortly. The third rule in Definition 1 is recursive, and it is necessary to implement the correct condition of execution of the basic blocks. This rule is related to how inter-thread control dependencies are implemented by MTCG, as described Step 3.

Consider the example in Figure 26. For thread $T_1 = \{A, B, C, J\}$, besides the start block $BB1$ and the end block $BB7$, the only relevant basic blocks are $BB2$, $BB3$, and $BB6$. All these blocks are relevant to T_1 by rule 1 in Definition 1, since there is no incoming dependence into this thread. For thread $T_2 = \{D, E, F, G, H\}$, besides the start and end blocks, several other basic blocks are relevant for different reasons. $BB4$ and $BB5$ contain instructions in T_2, and therefore are relevant by rule 1. Because of the dependencies from T_1 to T_2, with source instructions B and C (Figure 1(c)), the basic blocks that contain these instructions ($BB2$ and $BB3$) are relevant to T_2 by rule 2. Again, no basic block is relevant due to rule 3 in this case.

4.6.7.3 STEP 2: MOVING INSTRUCTIONS TO NEW CFGS

The second step of MTCG (lines 4 to 7) is to insert the instructions in T_i into their corresponding basic blocks in CFG_i. MTCG inserts the instructions in the same relative order as in the original code, so that intra-thread dependencies are naturally respected. Figure 26(d) and Figure 26(e) illustrate the instructions inserted in their threads, in the basic blocks corresponding to the instructions' original basic blocks.

4.6.7.4 STEP 3: INSERTING INTER-THREAD COMMUNICATION

The third step of the algorithm (lines 8 to 44) is to insert communication primitives to implement each inter-thread dependence. We represent communication primitives by **`produce`** and **`consume`** instructions. The **`produce`** and **`consume`** instructions are inserted in pairs, so that, statically, each **`produce`** feeds one **`consume`**, and each **`consume`** is fed by one **`produce`**. For each inter-thread dependence, a separate communication queue is used. Although a single queue per thread pair is sufficient, multiple per-dependence queues are actually used to maximize the freedom of subsequent single-threaded scheduling. Furthermore, a queue-allocation technique can later reduce the number of queues necessary, analogously to register allocation. Depending on where the communication primitives are inserted with respect to the parallel region, they can be of three types: *intra-region* (i.e. inside the parallelized region), *initial*, or *final* communications. We first describe the insertion of the most interesting ones, the intra-region communications, followed by the initial and final communications.

4.6.7.5 Intra-Region Communications

In order to preserve the conditions under which each dependence occurs, MTCG adopts the following strategy.

PROPERTY 3: BASIC COMMUNICATION PLACEMENT

Each intra-region dependence is communicated at the program point of its source instruction.

Figure 28. Property 3: Basic Communication Placement

This strategy is simple to implement, and it also simplifies the proof of correctness of the algorithm. However, this strategy can result in a suboptimal number of inter-thread communications. Optimizations to reduce inter-thread communications are described in 4.6.7.9.

In Figure 25, **add_before**($\boldsymbol{CFG_i}$**, I, instr)** inserts **instr** in $\boldsymbol{CFG_i}$ at the point right before instruction I's position in the original CFG, and add after works analogously. The **add_last(B, I)** function inserts instruction I as the last instruction in basic block B. The actual communication instructions inserted depend on the type of the dependence, as illustrated in Figure 26:

Register Dependencies: These are implemented by communicating the register in question.

Memory Dependencies: For these, purely synchronization primitives are inserted to enforce that their relative order of execution is preserved. This is dependent on the target processor's memory consistency model. In Figure 25, synchronization primitives are represented by **produce.rel** and **consume.acq** instructions, which implement the release and acquire semantics in the memory system.

Control Dependencies: In the source thread, before the branch is executed, its register operands are sent. In the target thread, consume instructions are inserted to get the corresponding register values, and then an equivalent branch instruction is inserted to mimic the same control behavior. For simplicity, branches are assumed to have a single register operand.

In the example from Figure 25, there are two inter-thread, intra-region dependencies. First, instruction C in T_1 writes into register $r2$ that is used by instructions D, F, and H in T_2. This register data dependence is communicated through queue 2. The **produce** and **consume** instructions are inserted right after the points corresponding to instruction C in $BB3$. The other inter-thread dependence is a control dependence from branch B in T_1, which controls instructions in T_2. This dependence is satisfied through queue 1. In T_1, the register operand ($p1$) is sent immediately before instruction B. In T_2, the value of $p1$ is consumed and a copy of branch B (B') is executed.

As mentioned in line 9 of figure Figure 25, the MTCG algorithm also needs to communicate transitive control dependencies in the PDG in order to implement the relevant blocks for a thread. This is related to rule 3 in Definition 1. The reason for this is control dependencies' implementation, which consumes the branch operands and duplicates the branch in the target thread. In order to enforce the correct conditions of execution of these instructions in the target thread (i.e. to create the relevant basic blocks to contain these instructions), it is necessary to communicate the branches controlling the blocks where these instructions are inserted. This reasoning follows recursively, thus resulting in the necessity of communicating the transitive control dependencies.

4.6.7.6 Initial Communications

Communication primitives may need to be inserted for register values that are live at the entry of the parallelized region, similarly to OpenMP's **`firstprivate`**. More specifically, if a register definition from outside the parallelized region reaches a use in an instruction assigned to thread T_i, then the value of this register has to be communicated from the main thread to T_i. These communication primitives are inserted at the $START$ basic block in CFG and at its corresponding $START_i$ basic block in CFG_i. This step is illustrated in lines 31-35 in Figure 25.

In the example from Figure 26, the main thread executes the code for T_1, so no initial communication is necessary for T_1. For T_2, instruction G uses the value of $r10$ defined outside the parallelized region, and therefore this register needs to be communicated. This dependence is communicated through queue 0, and the **`produce`** and **`consume`** instructions are inserted in $BB1$, the $START$ basic block.

Since auxiliary threads are spawned at the entry of the parallelized region, no initial communication for memory and control dependencies are necessary. In other words, auxiliary threads are only spawned when the execution reaches the parallelized region, and any memory instruction before this region will have already executed.

4.6.7.7 Final Communications

At the end of the parallelized region, the auxiliary threads may have to communicate back to the main thread, which is similar to OpenMP's **`lastprivate`**. Final communications may be necessary not only for register values, but also for memory. We discuss these two cases below.

For every register *r* defined by an instruction assigned to thread T_i, *r* needs to be sent to the main thread if two conditions hold. First, *r* must be live at the end of the parallelized region (END). Second, a definition of *r* in T_i must reach END. Lines 36-40 in Figure 7.9 handle this case.

In Figure 26, it is necessary to communicate the value of register $r10$ from T_2 to the main thread at the end of the parallelized region. This communication uses queue 3, and the **`produce`** and **`consume`** instructions are inserted in the END blocks (corresponding to $BB7$).

A problem arises in case multiple threads have definitions of *r* that reach the exit of the parallelized region. Depending on the flow of control, a different thread will have the correct/latest value of *r*, which needs to be communicated back to the main thread. To address this problem, two solutions are possible. One alternative is to associate a time-stamp for *r* in each thread, so that it is possible to tell which thread wrote *r* last. In this scheme, at the end of the parallelized region, all possible writers of *r* communicate both their value of *r* and the corresponding time-stamp to the main thread. The main thread can then compare the time-stamps to determine which value of *r* is the latest. The time-stamps can be computed based on a combination of topological order and loop iterations. An alternate solution is to enforce that a single thread always has the latest value of *r*. This can be enforced in two ways. First, the possible partitions of the PDG can be restricted to require that all instructions writing into *r* must be assigned to the same thread. Unfortunately, this restricts the valid partitions, breaking both the generality of the code generation algorithm and the separation of concerns between thread partitioning and code generation. A better alternative is to include in the PDG output dependencies among instructions defining the same live-out register *r*. Effectively, these dependencies will be communicated inside the parallelized region, similarly to true register dependencies. These register output dependencies need not be inserted among every pair of instructions defining a register *r*. Instead, one of these instructions (say, I) can be chosen, so that the thread containing I will hold the latest value of *r*. Register output dependencies can then be inserted from every other instruction writing into *r* to instruction I. This way, a final communication of *r* is necessary only from the thread containing I to the main thread.

At the exit of the parallelized region, synchronizations may also be necessary to enforce memory dependencies (lines 41-43 in Figure 25). Specifically, for every auxiliary thread that contains a memory instruction, it is necessary to insert a synchronization from it to the main thread upon the exit of the parallelized region. This synchronization prevents the main thread from executing conflicting memory instructions that may exist in the code following the parallelized region. Analyses can be performed to determine if such synchronization is unnecessary, or to move it as late as possible in the code.

4.6.7.8 STEP 4: CREATING NEW CFGS' ARCS

The last step of the MTCG algorithm (lines 45-53) is to create the correct control flow in the new CFGs. To achieve this, it is necessary both to adjust the branch targets and to insert jumps in these CFGs. Because not all the basic blocks in the original CFG have a corresponding block in each new CFG, finding the correct branch/jump targets is nontrivial. The key property we want to guarantee is that each basic block in a new CFG must have the same condition of execution of its corresponding basic block in the original CFG. In other words, the control dependencies among the basic blocks in a new CFG must mimic the control dependencies among their corresponding basic blocks in the original CFG.

Since control dependencies are defined based on the post-dominance relation among basic blocks, it suffices to preserve the post-dominance relation among basic blocks. Therefore, for a new CFG, CFG_i, the branch/jump targets are set to the closest post-dominator basic block B of the original target/successor in the original CFG such that B is relevant to CFG_i. Notice that such post-dominator basic block always exists as every vertex is post-dominated by END, which is relevant to every new CFG. Demonstrating that the control dependencies among the new basic blocks match the control dependencies among their corresponding blocks in the original CFG is central in the correctness proof of the MTCG algorithm. For simplicity, the MTCG algorithm inserts unconditional jumps at the end of every new basic block. Many of these jumps, however, can be later optimized by standard code layout optimizations.

Figure 26(d) and Figure 26(e) illustrate the resulting code, with the control-flow arcs inserted by MTCG. Determining most of the control-flow arcs is trivial in this example. An interesting case occurs in CFG_1 for the outgoing arc of BB3'. In the original CFG (Figure 26(b)), control flows from BB3 to BB4. However, BB4 is not relevant to T_1. Therefore, the output arc from BB3' goes to the closest post-dominator of BB4 relevant to T_2, which is BB6'. Another interesting case happens for branch E in T_2. In the original code, the branch target of E is BB6. However, BB6 is not relevant to CFG_2. Therefore, branch E is redirected to BB6's closest post-dominator relevant to T_2, which is BB2".

The code in Figure 26(d) and Figure 26(e) has been subject to code layout optimizations. For example, no jump is necessary at the end of BB3' if we place BB6' immediately after BB3' in the generated code.

4.6.7.9 Inter-Thread Communication Optimizations

Threads created by MTCG must synchronize via communication queues whenever a dependence edge is split across different threads. Though one of the benefits of pipelined parallelism is that performance is not dependent on cross-thread communication latency, excess communication can still hurt performance for other reasons. Among these are limited queue bandwidth and the cost of doing the produce or consume operation itself.

In the baseline MTCG algorithm, each parallel thread must maintain a control flow structure that corresponds to a subset of the control flow of the original code. When an earlier stage executes a branch that might affect the control flow of a later stage, the branch condition must be communicated to the later stage. Depending on the application and the partition, the need to respect these control dependencies may result in large numbers of dynamic communications between threads. For instance, consider an outer loop L containing two separate inner loops, A and B, where loop A contains an early exit. Loop A is assigned to stage 1 and loop B is assigned to stage 2. In this case, even though stage 2 only needs to perform the computation of loop B, it must receive the condition of the loop-back branch of loop A for each iteration of A, because it must know whether or not to execute loop B. Thus, stage 2 will contain two loops: a "skeleton" loop that only consumes the condition from stage 1, and the actual work in loop B.

The communication optimization phase of MTCG removes these unnecessary communications by transforming the control flow of the threads. In this particular example, the skeleton loop in thread 2 will be replaced by a single consume operation; the produce operation inside loop A in thread 1 will be replaced by a single produce operation after loop A in thread 1 that tells thread 2 whether or not to execute the loop. More generally, the MTCG communication optimization phase looks for groups of skeleton basic blocks that exist only to maintain consistent control flow, and simplifies them to reduce cross-thread communication.

4.6.8 Queue Implementation

Core-to-core communication bandwidth is critical for high-performance pipeline-parallel programs. For commodity multicore processors without hardware communication queues, (PS-)DSWP uses software queues to send and receive data values and synchronization tokens across pipeline stages. Although (PS-)DSWP is highly tolerant of communication latency, high-bandwidth low-overhead software queues are necessary to ensure high utilization of parallel hardware resources.

To address this issue, we have implemented Liberty Queues, a high-performance lock-free software-only ring buffer. Although Liberty Queues are currently implemented on x86-64 and IA-64 platforms [JZJ+10], many queue optimization techniques introduced in this section are applicable to other platforms as well. Liberty Queues achieve high bandwidth core-to-core communication on a wide range of multicore and multiprocessor systems. According to our experimental results on seven systems, including desktop and server microprocessors, the maximum core-to-core average bandwidth was 2.49 GB/s on a 2.4 GHz Xeon E5530, with an overhead of 3.21 ns per produce-consume pair. The minimum core-to-core average bandwidth was 281 MB/s on an Itanium 2, with an overhead of 15.9 ns. These performance results compare favorably with a recent proposal, FastForward, which achieved 281 MB/s with 28.5-31 ns overhead on a 2.66 GHz Opteron 2218 (Giacomoni, Moseley and Vachharajani 2008).

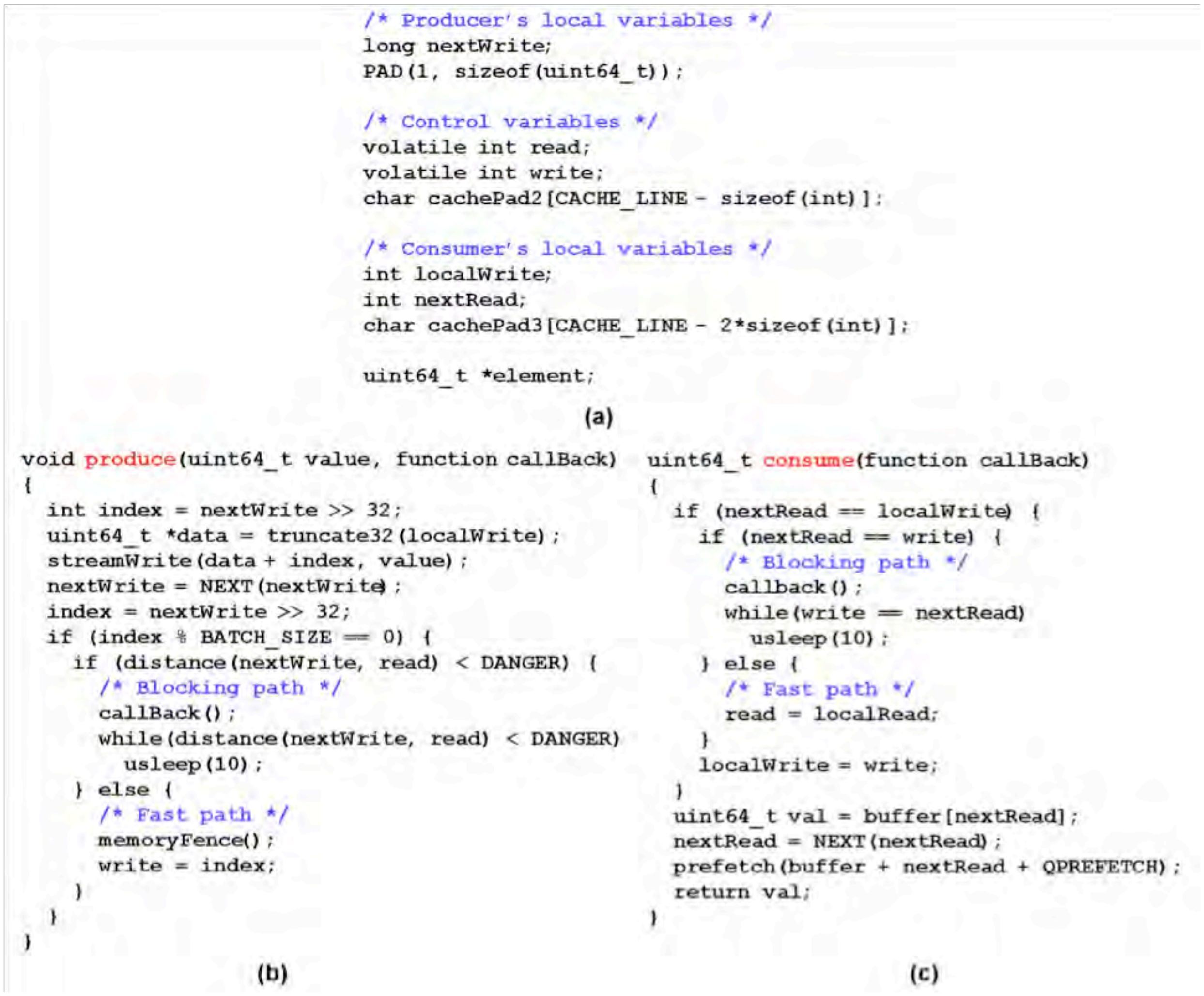
Liberty Queues have the following features:

Lock-free batch updates: Liberty Queues use a variation of the MCRingBuffer system of batch updates (Lee, Bu and Chandranmenon 2010) to reduce the frequency of updates of shared variables. Liberty Queues does not require the use of a lock to serialize producer and consumer.

Cache-aware data layout: Liberty Queues carefully lay out shared data structures to minimize cache line thrashing caused by ping-ponging shared variables between a pair of producer and consumer.

Exploiting asymmetry for producer-consumer load balancing: Unlike MCRingBuffer, where the control variables are owned equally by producer and consumer, Liberty Queues enable flexible division of communication overhead between a pair of producer and consumer. For example, in Software Multithreaded Transactions (SMTX) (Raman, et al. 2010), a form of speculative pipeline parallelism, the producer thread is always busier than consumer thread, therefore Liberty Queues shift the communication burden to the consumer thread.

Platform-specific optimization: Liberty Queues utilize platform-specific streaming store (e.g., **`movntiq`** in x86-64) and prefetching (e.g., **`prefetch.nta`** in x86-64) instructions for better bandwidth and performance stability. Streaming stores issued by a producer bypass L2 cache as soon as an entire cache line is written and violate x86's usual coherence guarantees. A consumer prefetches many accesses ahead using the 64-bit SSE prefetching mechanism, which improves bandwidth and performance stability.



```
/* Producer's local variables */
long nextWrite;
PAD(1, sizeof(uint64_t));

/* Control variables */
volatile int read;
volatile int write;
char cachePad2[CACHE_LINE - sizeof(int)];

/* Consumer's local variables */
int localWrite;
int nextRead;
char cachePad3[CACHE_LINE - 2*sizeof(int)];

uint64_t *element;
```

```
void produce(uint64_t value, function callBack)
{
  int index = nextWrite >> 32;
  uint64_t *data = truncate32(localWrite);
  streamWrite(data + index, value);
  nextWrite = NEXT(nextWrite);
  index = nextWrite >> 32;
  if (index % BATCH_SIZE == 0) {
    if (distance(nextWrite, read) < DANGER) {
      /* Blocking path */
      callBack();
      while(distance(nextWrite, read) < DANGER)
        usleep(10);
    } else {
      /* Fast path */
      memoryFence();
      write = index;
    }
  }
}
```

```
uint64_t consume(function callBack)
{
  if (nextRead == localWrite) {
    if (nextRead == write) {
      /* Blocking path */
      callback();
      while(write == nextRead)
        usleep(10);
    } else {
      /* Fast path */
      read = localRead;
    }
    localWrite = write;
  }
  uint64_t val = buffer[nextRead];
  nextRead = NEXT(nextRead);
  prefetch(buffer + nextRead + QPREFETCH);
  return val;
}
```

Figure 29. Liberty Queues

A simplified version of the Liberty Queues. (a) The Liberty Queue internal data structure; (b) The Liberty Queue Produce Process; (c) The Liberty Queue Consume Process

Figure 29 shows a simplified version Liberty Queues. Figure 29(a) shows the Liberty Queue data structure with three sections. The first section contains the producer's local variables. These variables are read and written by the producer and never the consumer. The next section contains control or global variables read and written by producers and consumers. The last section contains the consumer's local variables, which are read and written only by the consumer.

Figure 29(b) shows the Liberty Queue produce function. To produce a value, the Liberty Queue first writes a value to the queue. If this produce completes a batch of data, check to see if the producer is in danger of overtaking the consumer. If so, execute a callback and sleep, waiting for the situation to improve. If not, update the control variables to make writes visible to the consumer. Updating the control variables to make writes visible to the consumer is called flushing the queues and is analogous to flushing a file. For correctness, the callback function must flush the queue that called it. Liberty Queues utilize Streaming Single Instruction Multiple Data (SIMD) Extension's (SSE) streaming store instruction for better bandwidth and performance stability. Streaming stores bypass L2 cache as soon as an entire cache line is written and violate x86's usual coherence guarantees. Loads will not see the values of streaming stores until `sfence` instruction.

In the fast common case, the non-inlined version of Liberty Queues require only one load and two stores per invocation. Liberty Queues sacrifice readability and flexibility for raw performance. In particular, Liberty Queues make batch size a compile time decision and require it to be a power of two for best performance. Additionally, Liberty Queues always allocate the underlying queue buffer somewhere in the lower 32 bits of memory. The lower 32-bits of `nextWrite` hold the buffer's address, and the upper 32-bits hold the index into the buffer. When inlining, GCC is clever enough to promote `nextWrite` to a register. Inlined Liberty Queues require zero loads and one store per produce.

Figure 29(c) shows the Liberty Queue consume function. When consuming, the Liberty Queue first checks the local copy of the write variable to determine if any values are available to dequeue. If not, the queue consults the shared control variables. If this check fails as well, the callback function executes and the queue sleeps, waiting for the producer to enqueue more data. If the control variable indicates there are more values already enqueued, the consumer updates the read control variable with its copy of local state. Whether or not the queue blocked, the consumer updates the local copy of the write state. Copying the local read state to the read control variable is called reverse flushing. A correct callback function for the consumer must at least reverse flush the queue. At this point, regardless of the path of execution, there is at least one value ready to dequeue. The consumer dequeues a value and updates local state. Finally, the consumer prefetches a distant future value from the queue, and returns the dequeued value.

Note that in this implementation of Liberty Queues the consumer does not update the global read state unless it updates its local copy of the write state. This policy greatly favors producers over consumers. Every time the consumer writes to the control variables it will cause a cache ping-pong, so the consumer delays these writes as long as possible. The consumer prefetches many accesses ahead using the 64-bit SSE prefetching mechanism. Unlike prefetching with a load instruction, the SSE prefetch instruction will never cause a stall, does not require a register, and will not fault if the address is unreadable.

Liberty Queues use three major tunable parameters and benefit greatly from system characterization for performance tuning. More details will be discussed in the following section.

4.6.9 Exploiting System Characterization Parameters

The (PS-)DSWP parallelizer exposes many parameters towards seamless integration with system characterization to provide robust performance across a variety of hardware platforms and workloads. These parameters can be divided into communication queue parameters in section 4.6.9.1 and non-communication parameters in section 4.6.9.2.

4.6.9.1 Communication Queue Parameters

Liberty Queues use three major tunable parameters, all of which are functions of lower-level hardware parameters such as cache line size, cache size, synchronization cost, inter-core communication bandwidth, TLB size, prefetcher design, and so on:

Queue size: Larger queue size helps compensate for bursty communication patterns or consumers with variable processing times. Experiments showed that the only disadvantage of long queues was potentially exhausting 32-bit address space. Once the queues are long enough to buffer out bursts by producers and slow processing by consumers, there are no performance gains for increasing the size further. Liberty Queues default to 16 megabytes.

Batch size: Batch size determines how often the producer will update global state. Larger batches mean fewer updates and better performance, but updating too infrequently causes the consumer to spin wait uselessly. Batch size defaults to 128 kilobytes.

Prefetch distance: Prefetch distance determines how early the consumer will read data. The idea is to request data far enough in advance that it reaches the L2 cache before a real request for that data. However, reading data too early can cause cache pollution. Liberty queues prefetch 1 kilobyte ahead.

From the operational standpoint, the AESOP compiler team provides a parameterized version of Liberty Queues, and the AESOP characterization team performs system characterization to find an optimal set of queue parameters for a given hardware platform.

4.6.9.2 Non-Communication Parameters

The balance of pipeline is crucial for optimal performance and affected by interplay of multiple architectural parameters. Important system characterization parameters for DSWP and PS-DSWP include:

Number of Effective Parallel Contexts: The number of effective parallel contexts directly affects the shape of pipeline such as the number of pipeline stages and the number of allocated parallel contexts per each stage.

Execution cost of (a block of) instructions: The partitioning algorithm of (PS)DSWP counts on the estimation of execution time of a code block to balance pipeline stages. Currently, a relatively simple cost model is being used; the cost of a code block is a sum of the costs of individual instructions in the code block, where each type of instructions (e.g., arithmetic, branch, load and store, floating-point arithmetic) has a fixed cost. In the future, a better cost model based on system characterization would be ideal, which calculates the cost of an arbitrary block of code.

5.0 Conclusions

Our assessment is that the problem statement as captured here and in the original DARPA BAA is correct. If anything, the time that has passed since the original BAA has proven the problem is worse than originally characterized. For example, the push for GPUs to become GP-GPUs and the continued use of FPGAs—which are exceedingly difficult to program—shows that the need for high performance parallel hardware is making design teams go to extreme lengths. And yet, the ease of writing applications that make effective use of all the parallel resources available to them has not improved. The crisis DARPA—and many in the commercial sector—has called out is real, is pressing and is not going away. The approach taken by the AESOP team to attack this problem through an automatically adapting and parallelizing compiler appears very effective based on early results.

6.0 Recommendations

We believe DARPA should resurrect this program. This should happen quickly but in a different form. Instead of a program tucked under a High-Performance Computing area, a full parallel programming program should be a major DARPA program. A full complement of research into compilers, languages, debuggers, complete IDEs and even training should be created to help address this identified and critical parallel computing crisis. Until then this crisis will continue to affect military systems detrimentally until it is thoroughly addressed.

BIBLIOGRAPHY

Agakov, F., et al. "Using maching learning to focus iterative optimization." *Proceedings of the International Symposium on Code Generation and Optimization.* IEEE Computer Society, 2006. 295-305.

Allen, R., and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

Amarasinghe, S. P., Anderson, J. M., Lam, M. S., and Lim, A. W. "An Overview of a Compiler for Scalable Parallel Machines." Edited by D. Gelernter, A. Nicolau, and D. A. Padua U. Banerjee. *Proceedings of the 6th international Workshop on Languages and Compilers For Parallel Computing.* London: Springer-Verlag, 1994. 253-272.

Appel, Andrew W. and Maia Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

August, David I., et al. "Automatic Extraction of Parallelism from Sequential Code." In *Fundamentals of Multicore Software Development*, by Ali-Reza Adl-Tabatabai, Victor Pankratius and Walter Tichy. Chapman & Hall / CRC Press, 2011.

Bacon, David F., Susan L. Graham, and Oliver J. Sharp. "Compiler transformations for high-performance computing." *ACM Computing Surveys* 26, no. 4 (Dec 1994): 345 - 420.

Banerjee, P., Chandy, J. A., Gupta, M., Hodges IV, E. W., Holm, J. G., Lain, A., Palermo, D. J., Ramaswamy, S., and Su, E. "The Paradigm Compiler for Distributed-Memory Multicomputers." *Computer* 28, no. 10 (Oct 1995): 37-47.

Banerjee, U. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

Banerjee, U. "Speedup of Ordinary Programs." PhD thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, 1979.

Blume, W., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D. A., Petersen, P., Pottenger, W. M., Rauchwerger, L., Tu, P., and Weatherford, S. "Polaris: Improving the Effectiveness of Parallelizing Compilers." Edited by U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua K. Pingali. *Proceedings of the 7th international Workshop on Languages and Compilers For Parallel Computing.* London: Springer-Verlag, 1995. 141-154.

Cooper, K. D., et al. "ACME: Adaptive compilation made efficient." *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems.* ACM, 2005. 69-77.

Duchateau, Alexandre, Albert Sidelnik, María Jesús Garzarán, and David Padua. "P-RAY: A Suite of Micro-benchmarks for Multi-core Architectures." *Proc. of the International Workshop on Languages and Compilers for Parallel Computing.* 2008.

Eigenmann, R. , J. Hoeflinger, Z. Li, and D. Padua. "Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs." *Proc. Fourth Workshop on Languages and Compilers for Parallel Computing.* Santa Clara, 1991. 65–83.

Giacomoni, J., T. Moseley, and M. Vachharajani. "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue." *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 2008.

Goff, G., Kennedy, K., and Tseng, C. "Practical dependence testing." *SIGPLAN* (ACM) 26, no. 6 (June 1991): 15-29.

Gonzalex-Domingues, J., G. Taboada, B. Fraquela, M. Martin, and J. Tourino. "Servet: A Benchmark Suite for Autotuning on Multicore Clusters." *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium.* Atlanta: IEEE, 2010.

Harris, William R, Franjo Ivancic, Aarti Gupta, and Sriram Sankaranarayanan. "Program analysis via satisfiability modulo path programs." *ACM SIGPLAN Notices*, 2010: 71-82.

Jablin, Thomas B., Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. "Automatic CPU-GPU Communication Management and Optimization." *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, June 2011.

Lee, Patrick, Tian Bu, and Girish Chandranmenon. "A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring." *Proceedings of the 24th International Parallel and Distributed Processing Symposium.* Atlanta, 2010.

Liao, S.-W., A. Diwan, A. M. Ghuloum, and M. S. Lam. "SUIF explorer: An interactive and interprocedural parallelizer." *Principles Practice of Parallel Programming,* 1999: 37-48.

McKinley, K S. "A Compiler Optimization Algorithm for Shared-Memory Multiprocessors." *IEEE Transactions on Parallel and Distributed Systems* 9, no. 8 (Aug 1998).

Polychronopoulos, C. D., Gikar, M. B., Haghighat, M. R., Lee, C. L., Leung, B. P., and Schouten, D. A. "The structure of parafrase-2: an advanced parallelizing compiler for C and FORTRAN." Edited by A. Nicolau, and D. Padua D. Gelernter. *Selected Papers of the Second Workshop on Languages and Compilers For Parallel Computing.* Urbana, Illinois: Pitman Publishing, London, 1990. 423-453.

Raman, A., H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. "Speculative Parallelization Using Software Multi-threaded Transactions." *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems.* Pittsburgh, 2010.

Sussman, Alan, Norman Lo, and Tim Anderson. "Automatic System Characterization for a Parallelizing Compiler." *Proceedings of the IEEE Cluster 2011 Conference.* IEEE, 2011.

Tate, R., M. Stepp, Z. Tatlock, and S. Lerner. "Equality saturation: A new approach to optimization." *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 2010.

Teodoro, George, and Alan Sussman. "AARTS: Low Overhead Online Adaptive Auto-tuning." *Proceedings of ACM SIGPLAN 1st International Workshop on Adaptive Self_Tuning Computing Systems for the Exaflop Era.* ACM, 2011.

Tiwari, Ananta, Vahid Tabatabaee and Jeffrey Hollingsworth. "Tuning Parallel Applications in Parallel." *Parallel Computing* 35, no. 8-9 (August 2009).

Triantafyllis, S., N. Vachharajani, and D. I. August. "Compiler optimization-space exploration." *Proceedings of the '03 International Symposium on Code Generation and Optimization.* 2003. 204-215.

Vandierendonck, H., S. Rul, and K. De Bosschere. "The Paralax Infrastructure: automatic parallelization with a helping hand." *Proceedings of the 19th international conference on Parallel architectures and compilation techniques.* ACM, 2010. 389-400.

Wolf, M. E. and M. S. Lam. "A Loop Transformation Theory and An Algorithm to Maximize Parallelism." *IEEE Transactions on Parallel and Distributed Systems*, Oct 1991: 452–471.

Wolf, M., D. Maydan, and D. Chen. "Combining loop transformations considering caches and scheduling." *Proceedings of the 29th Annual International Symposium on Microarchitecture.* 1996. 274-286.

Wolfe, M. "How compilers and tools differ for embedded systems." *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems.* New York: ACM, 2005.

Wolfe, Michael. *Optimizing Supercompilers for Supercomputers.* London: Pitman Publishing, 1989.

Yellareddy, Greeshma, Aparna Kotha, Mitchell Katz, and Rajeev Barua. "An Integrated Program Representation for Loop Optimizations." Computer Science, University of Maryland, 2011.

Yotov, Kamen, Keshav Pingali, Paul Stodghill. Siebel Center for Computer Science. May 31, 2004. http://polaris.cs.uiuc.edu/~padua/cs426-03/Reference-3.pdf.

LIST OF ACRONYMS, ABBREVIATIONS, AND SYMBOLS

ACRONYM	DESCRIPTION
AACE	Architecture Aware Compiler Environment
AESOP	Adaptive Environment for Supercompiling with Optimized Parallelism
AFRL	Air Force Research Laboratory
AMD	Advanced Micro Devices, Inc.
ANSI	American National Standards Institute
API	Application Programming Interface
AST	Abstract Syntax Tree
BAA	Broad Agency Announcement
CDRL	Contract Data Requirements List
CFG	Control Flow Graph
CGS	Cache and Granularity Scheduler
CMT	Cyclic Multi-Threading
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
DOD	Department of Defense
DSWP	Decoupled Software Pipelining
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GP-GPU	General Purpose GPU
HCI	Human Compiler Interaction
IDE	Interactive Development Environment
IMT	Independent Multi-Threading
IR	Intermediate Representation
KB	Knowledge Base
LLVM	Low Level Virtual Machine
MIMD	Multiple Instruction Multiple Data
MMU	Memory Management Unit
MPI	Message Passing Interface
MTCG	Multi-Threaded Code Generation
LNT	Loop Nest Tree
NUMA	Non-Uniform Memory Access

ACRONYM	DESCRIPTION
OKB	Optimization Knowledge Base
OSAL	OS Adaptation Layer
PDG	Program Dependence Graph
PMA	Performance Model Analysis
PMT	Pipelined Multi-Threading
PS-DSWP	Parallel Stage Decoupled Software Pipelining
POSIX	Portable Operating System Interface for UniX
RISC	Reduced Instruction Set Computer
SCC	Strongly Connected Components
SDSC	San Diego Supercomputer Center
SIMD	Single Instruction Multiple Data
SMTAA	Satisfiability Modulo Theories Alias Analysis
SSA	Static Single Assignment
SSE	Streaming SIMD Extension
T2	Task Two (an evaluation team)
TLB	Translation Lookaside Buffer
TPP	Thread Partitioning Problem
UIUC	University of Illinois at Urbana-Champaign
UMA	Uniform Memory Access

INDEX

A

Active Harmony · 12–13
Adobe · 7
affine · 26
  array · 13
  parallelization · 38
Affine Parallelizer
  Algorithm for · 42
AMD · 9
API · 6, 75
Apple · 7
array privatization · 47
AST · 75
August, David · 15

B

BAA · 75
barrier
  modeling · 26
barriers · 44
Barua, Rajeev · 13
basic block · 29
benchmark · 26
benchmarks
  micro · 5, 13, 22, 25
  nano · 22, 25
  platform · 22
  Polybench · 48
blocking · *See* strip mining
broadcasts · 44

C

cache · 25
  L1 · 25, 28
  L2 · 25
  L3 · 25
Cache
  characteristics · 45
cache and granularity scheduler · *See* CGS
CDRL · 75
CFG · 29, 57, 75
CGS · 43, 75
characterization
  dynamic · 13–15
CMT · 16, 75
CMT+ · 43
communications
  final · 63
  initial · 63
  intra-region · 62
computation
  floating-point · 26
  integer · 26
  memory · 26
contexts · 25
  effective · 26
Control Flow Graph · *See* CFG
Control Speculation · 43
Cyclic Multi-Threading · *See* CMT

D

Decoupled Software Pipelining · *See* DSWP
dependencies · 29
  control · 29, 62
  data · 29
  memory · 29, 62
  register · 29, 62
DSWP · 18, 37, 43, 75

E

edge removal · 31

F

FIFO · 26, 75

G

GCC · 9, 75
GLSL · 11, 75
GNU Compiler Collection · *See* GCC
GP-GPU · 75
GPU · 75

H

HCI · 32, 75
heuristic
  idempotency · 36
  optimistic limit · 36
  super-idempotency · 36
human-compiler interaction · *See* HCI
hyperthreading · 25

I

IMT · 15, 16, 75
Independent Multi-Threading · *See* IMT
Intermediate Representation · *See* IR
IR · 9, 14, 75
  processing · 40

K

KB · 75
Knowledge Base · 6

L

Lattner, Chris · 7
Liberty Queues · 66
  features of · 66
  figure of · 67
LLVM · 7–12, 75
  diagram of · 7
  passes · 11, 39
  plugin architecture · 11
LNT · 42, 75
loop
  fission · 46
  fusion · 46
  interchange · 46
  interleaving · 47
  reversal · 47
  skewing · 47
  splitting · 47
loop analysis · 31
Loop Nest Tree · *See* LNT
Low Level Virtual Machine · *See* LLVM

M

Mac OS X · 11
micro-benchmarks · 22
MIMD · 4, 75
MMU · 25, 75
MPI · 4, 75
MTCG · 30, 50, 75
  algorithm · 57–65
Multi Threaded Code Generation · *See* MTCG

N

nano-benchmarks · 22
Negotiator · 32–40
  example · 36
NUMA · 26, 75

O

OKB · 76
open source · 7
OpenGL · 11
OSAL · 76

P

page coloring · 25
parallelizer
  affine · 13–15
  Non-affine · 15–20
  software pipelining · 50
Parallel-Stage DSWP · *See* PS-DSWP
PDG · 11, 20, 29–31, 30–31, 42, 57, 76
Pipelined Multi-Threading · *See* PMT
PMA · 76
PMT · 17, 76
Polyhedral model · 44
POSIX · 25, 76
P-Ray · 12–13
Privatization · 56
Program Dependence Graph · *See* PDG
PS-DSWP · 19, 43, 68, 76

R

Reduction · 47
RISC · 9, 76

S

San Diego Supercomputer Center · *See* SDSC
Satisfiability Modulo Theories Alias Analysis · *See* SMTAA
SCC · 51, 76
SDSC · 22, 76
  Penguin · 27
Servet · 12–13, 25, 28
SIMD · 67, 76
SMTAA · 39, 76
software pipelining · 26
SSA · 9, 76
SSE · 67, 76
Static Single Assignment · *See* SSA
Streaming SIMD Extension · *See* SSE
strip mining · 48
Sussman, Alan · 13

T

T2 · 21, 76
Thread-Partitioning Problem · *See* TPP

TLB · 25, 76
TPP · 52, 76
translation lookaside buffer · *See* TLB

U

UMA · 25, 76
Unified Memory Access · *See* UMA

X

X-Ray · 12–13, 22, 23, 25, 28