Metaoptimization of the In-lining Priority Function for a Compiler Targeting a Polymorphous Computing Architecture

Laurence D. Merkle
Rose-Hulman Institute of Technology
5500 Wabash Ave., CM-103
Terre Haute, IN 47803
(812) 877-8474
l.merkle@ieee.org

ABSTRACT
Leading polymorphous computing architecture (PCA) efforts include the Raw Architecture Workstation (Raw) and the Tera-op Reliable and Intelligently Adaptive Processing System (TRIPS), both of which are tile-based. The Raw toolchain places responsibility for program decomposition on the programmer, but the TRIPS toolchain automatically generates hyperblocks and allocates them to processing elements. This report identifies evolutionary computation (EC) techniques that enable and that are enabled by PCA technology, focusing on application of EC in enhancing the effectiveness of the TRIPS toolchain, including the Scale compiler. In particular, computational experiments are described that investigate the application of genetic programming to the meta-optimization of the priority function used to increase the number of instructions per hyperblock in the in-lining optimization phase of Scale. Results suggest continued experimentation with larger population sizes and more generations.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *compilers, optimization.* I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search – *heuristic methods.*

General Terms
Algorithms, Performance, Experimentation.

Keywords
Evolutionary computation, polymorphous computing architectures, compiler optimization.

1. INTRODUCTION
Polymorphous computing architectures (PCAs) represent a revolutionary approach to computing systems that seeks to provide processing capabilities that are both amenable to dynamic optimization as the application load changes and scalable with technology advances. Leading efforts achieve dynamic responsiveness and scalability through the use of tile-based architectures of some variety. Two of those efforts have direct relevance to this effort: the Raw Architecture Workstation (Raw) and the Tera-op Reliable and Intelligently Adaptive Processing System (TRIPS).

There is considerable potential for even better performance through the development of architecture-specific optimizations. Evolutionary computation (EC) is a maturing field which centers on the study of evolutionary algorithms (EAs) – algorithms that are inspired by principles and theories of natural evolution – of which genetic algorithms are the best publicized example. This effort began the exploration of the use of EC techniques that enable and are enabled by PCA technology.

Because of their population-based nature, EC techniques are amenable to a rich variety of implementations on parallel and distributed architectures and scale very well with processor count. Much of the research in this area carries over directly to their implementation on tile-based PCAs. As such, EC techniques could be distributed spatially across a tile-based architecture to provide dynamic performance optimization. As a preliminary step towards this goal, island model and farming model parallel EA implementations for both the Raw and TRIPS architectures were designed and implemented.

The primary goal of the effort was to develop versions of both the Raw and TRIPS compilers that combine EC techniques for robust global search of the schedule space with the compilers' existing algorithms for efficient local search. However, in the process of developing the Raw implementations of the parallel EAs, it was determined that the Raw toolchain essentially treats each tile independently, thereby requiring the programmer to decompose the application and map the components to the tiles. Creating an automatic decomposition tool from scratch was not possible within the timeframe of this project, so subsequent effort was focused on the TRIPS architecture.

The most important technical accomplishments under this effort are as follows:

- Gained familiarity with Raw and TRIPS toolchains
- Implemented simple EAs for the Raw and TRIPS architectures, and evaluated the implementations using a Raw handheld board installed on a Linux workstation and TRIPS architecture simulator software
- Identified several classes of methods for the application of EC in enhancing the effectiveness of the Raw and TRIPS

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

toolchains; selected general method for use in enhancing TRIPS toolchain

- Installed the Finch Meta-Optimization Framework, which uses "machine learning techniques to automatically search for effective compiler heuristics."

- Obtained preliminary results in application of Finch to optimization of in-lining priority heuristic used by the TRIPS compiler (8% reduction in static ratio of hyberblocks to instructions per hyberblock)

- Verified through a single processor computational experiment on a nontrivial test case that parallel computation will be required in order to obtain meaningful data about the effectiveness of EA-based optimization of TRIPS compiler heuristics

- Determined that the Ground Moving Target Indicator (GMTI) benchmark provides a suitable test case for EA-based optimization of TRIPS compiler heuristics, in that the effectiveness of function in-lining varies gradually with the code bloat size parameter for this benchmark.

- Implemented a parallel version of Finch on a Beowulf cluster using the Message Passing Interface (MPI).

- Completed a 17-processor computational experiment applying Finch to the optimization of the TRIPS in-lining priority heuristic using the GMTI benchmark as the test case.

The remainder of this section provides background information on the most relevant aspects of polymorphous computing architectures, compilers, and evolutionary computation. Section 2 describes the methods used in this effort, and the results of the research are presented and discussed in Section 3. The remaining sections present conclusions (Section 4), recommendations (Section 5) and references (Section 6).

1.1 Polymorphous Computing Architectures

Current computing systems are designed to support fixed, idealized application loads, and their performance inevitably suffers when the actual load doesn't match the idealized load for which they were designed. Also, as manufacturing processes for integrated circuits advance and we approach the fundamental limitations of silicon technology, wire delays are becoming more significant relative to gate delays. PCAs represent a revolutionary approach to computing systems that seeks to provide processing capabilities that are both amenable to dynamic optimization as the application load changes and scalable with technology advances. Leading efforts in PCA research include the Raw microprocessor under development at the Massachusetts Institute of Technology and the TRIPS architecture project at the University of Texas at Austin. Both of these efforts achieve dynamic responsiveness and scalability through the use of tile-based architectures of some variety.

1.1.1 Raw

MIT researchers argue that we must reconsider our idea of machine instructions to include signal routing information along with the usual functional unit control information.[1] The Raw microprocessor makes this possible, and has been demonstrated to provide two orders of magnitude better performance than traditional processors on certain applications. However, optimization of the routing information places an additional burden on the compiler. Compiler enhancements implemented just prior to the initiation of this research resulted in code with speed and tile usage that typically come close to hand-customized code [10], but independent evaluations resulted in only two thirds of the theoretically possible efficiency, suggesting that further optimization is possible [9].

1.1.2 TRIPS

Researchers at the University of Texas Austin also suggest a new paradigm for machine instructions, illustrated by their TRIPS architecture [4]. They advocate the adoption of Explicit Data Graph Execution (EDGE) architectures, in which "the hardware delivers a producer instruction's output directly as an input to a consumer instruction," thereby eliminating most of the expensive logic that has found its way into architecture design over the past two decades. In addition to the usual requirements, such as identifying blocks of instructions containing no branches, a compiler targeting such an architecture must be able to map each such block to a tile for execution, and then map each operation in the block to a processing element. These spatial scheduling mappings affect both concurrency and communications delays, and thus result in a difficult multicriteria optimization problem. The greedy approximation algorithm employed by the TRIPS compiler prior to the initiation of this research results in code that is highly un-optimized and bloated, which suggests an opportunity to improve performance via a variety of optimization techniques [3].

1.2 Compilers

In theoretical terms, the role of a compiler is to automatically translate sentences from one language into sentences in another language. By far the most familiar use of compilers, and the one usually connoted by the use of that term, is in translating source code (sentences in high-level programming languages) into object code (sentences written in machine language, annotated to support linking and relocation). Although it is usually hidden from the user, it is common for the compiler to emit assembly language code and then invoke an assembler to make the final translation to object code.

Often the compilation process occurs on a computing system based on the targeted architecture, but that is not always the case. Cross compilers execute on one architecture and generate executable programs for a different architecture, and are usually the only option when targeting new architectures.

Implementations vary widely, but the translation process may be conceived of as performing a sequence of operations including lexical analysis, preprocessing, parsing, semantic analysis, optimization[1], and code emission. The first four of these result implicitly or explicitly in an intermediate representation that is independent of the target architecture, and as such are referred to as the front end. The back end generates the object code, usually after performing architecture-specific optimizations. Many compilers also perform architecture-independent optimizations, which can be thought of as occurring late in the front end, early in

[1] As stated in the classic compiler textbook by Aho, Sethi, and Ullman, [2] "the term 'optimization' is a misnomer because there is rarely a guarantee that the resulting code is the best possible." In a nutshell, this is the precise reason to consider the application of evolutionary algorithms as optimum-seeking techniques in the compilation process.

the back end, in a separate middle end, or in some combination thereof.

1.2.1 In-lining

This research is primarily concerned with the optimization aspects of compilation, regardless of the end in which they occur and whether or not they are architecture-specific. Computational experiments so far have focused on in-lining, which is the substitution of the body of a function for a function invocation. In-lining can improve the efficiency of the resulting executable by eliminating the overhead of the function invocation. Furthermore, because it simplifies control flow, it results in larger basic blocks, which is a major consideration in generating efficient executables for the TRIPS architecture.

The advantages of in-lining must be traded off against the resulting bloat in the executable code. One approach for managing this tradeoff is to constrain the code bloat to a user-specifiable fraction of the code size in the absence of in-lining. This approach assigns the compiler the responsibility of choosing a subset of functions to in-line. The underlying subset selection problem is NP-complete, calling for an approximation algorithm or heuristic solution. The standard solution is to rank the available functions on the basis of a heuristic priority function, and then in-line as many of the most highly ranked functions as possible without exceeding the code bloat constraint.

1.2.2 Scale

The TRIPS project uses the Scalable Compiler for Analytical Experiments (Scale) compilation system, which is intended as a research and instructional tool to support the development of more powerful, flexible, and reusable compilers [11]. Conversely, the Scale project currently focuses on the generation of high performance code for the TRIPS architecture, which it does through the use of advanced in-lining and predicated loop unrolling techniques.

Scale considers each routine invocation for in-lining independently, as opposed to considering all invocations of a given routine together. The priority function used is the ratio of the execution frequency of the basic block containing the invocation to the size of the routine. The execution frequency of the block is estimated using its nesting level.

Features currently implemented by the Scale system include "parsers for C, Fortran, and Java byte-codes, alias analyses, static single assignment form (SSA), a collection of scalar optimizations, (PRE, value numbering, copy propagation, dead code elimination, and constant propagation)."

The high-level data flow aspects of the Scale compilation system are shown in Figure 1. The parser generates an Abstract Syntax Tree (AST) expressed in Clef [13], which is then converted to a Control Flow Graph (CFG). Certain optimizations, including in-lining, are applied to this representation. The CFG is then converted to Static Single Assignment (SSA) form before additional optimizations are applied, including loop unrolling.

1.3 Evolutionary Computation

EC is a maturing field which centers on the study of algorithms that are inspired by principles and theories of natural evolution. Exploiting the analogy to the principle of "survival of the fittest," EAs have been used in a wide variety of both static and dynamic optimization problems. In this role, they have been observed to be more scalable with respect to problem size than other global optimization techniques. More generally, in analogy to the processes by which whole species adapt to changing environments, EAs enable software to adapt to dynamic changes in the execution environment.

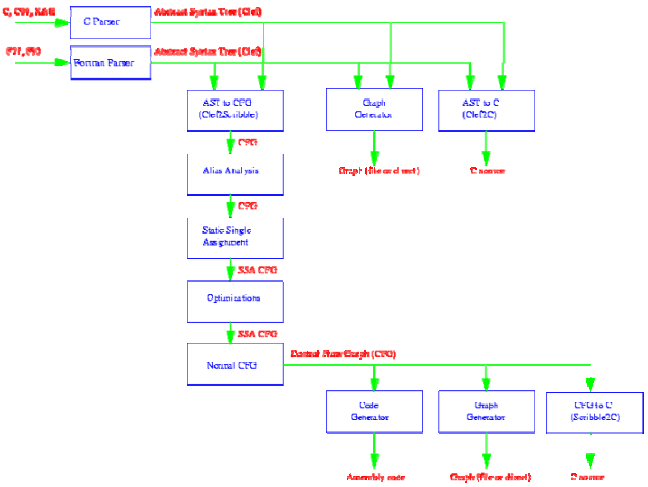

Figure 1. Scale Data Flow Diagram (http://www-ali.cs.umass.edu/Scale)

1.3.1 Genetic Programming

Genetic programming (GP) is a form of evolutionary computation in which the individuals evolved are algorithms. The standard representation is Lisp-like [7]. Specifically, algorithms are represented as trees, with each internal node representing an operator and each leaf representing an operand. This allows efficient evaluation of each candidate algorithm while still facilitating effective search of the space of algorithms.

The standard recombination operator randomly selects a node within each individual and exchanges the subtrees rooted at those nodes. Various forms of mutation are possible, including replacing a randomly selected subtree with a new randomly generated one.

1.3.2 Parallel EAs

Because EAs are population-based, there are a number of reasonable models for the implementation of EAs on parallel architectures [5]. Of these models, the two that are relevant to this effort are the island model and the farming model. Under the island model, the population is decomposed into subpopulations, each of which is allocated to a processing element. The subpopulations evolve independently, except that individuals occasionally migrate between populations. In a farming model EA, evolution takes place on a single processor, except that fitness evaluations are performed by the remaining processors in the system.

1.4 Finch

As described above in the context of in-lining, compiler optimizations often involve underlying NP-complete problems calling for approximation algorithms or heuristic solutions. The development of heuristics that are effective across a set of applications requires considerable effort. Historically, heuristics have been tuned carefully for the architectures in use at the time a compiler is initially developed and then left unmodified as the target architectures evolve and become more complex, resulting in decreased effectiveness.

In order to address this problem, researchers at MIT have developed GP-based Meta-optimization software (a.k.a. Finch) to automatically search the space of compiler heuristics [12]. The system specifically targets those heuristics that are based on priority functions such as the one described above in the context of in-lining.

The flow of execution in an application of Finch to the optimization of a compiler as it pertains to this effort is illustrated in Figure 2. At a high level of abstraction, Finch executes a standard evolutionary algorithm consisting of the steps labeled GP Initialization, Fitness Evaluation, Evolution (application of evolutionary operators), and GP Finalization. The individuals being evolved are heuristic functions, and their evaluation is performed by invoking a modified version of the compiler.

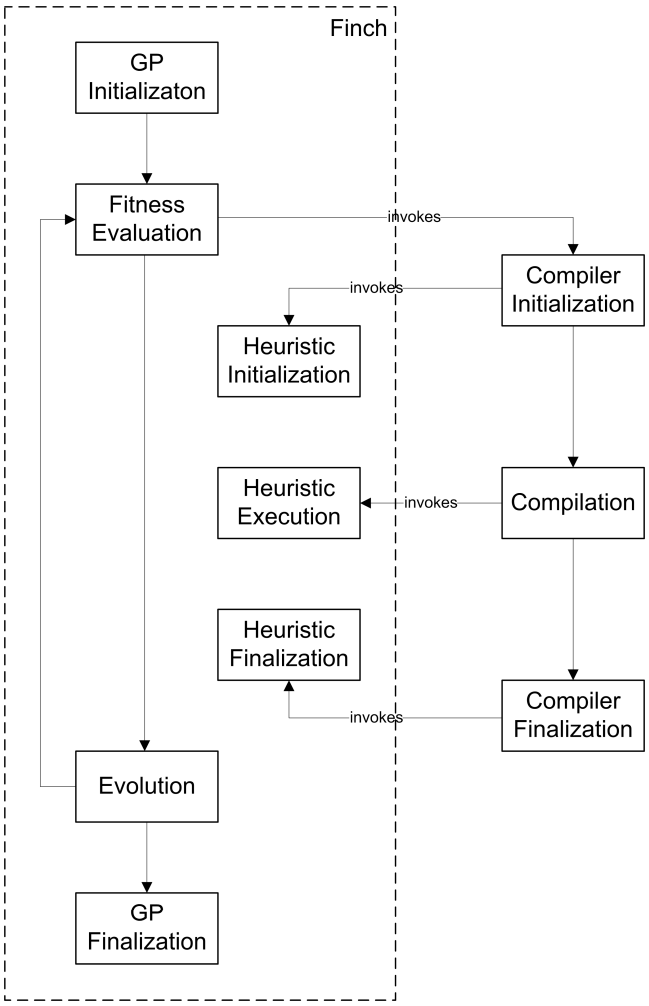

Figure 2. Finch Flow of Execution

During its own Compiler Initialization and Compiler Finalization steps, the compiler performs calls back to Finch to initialize and finalize its evaluation of the specific heuristic function under consideration (Heuristic Initialization and Heuristic Finalization, respectively). The primary purpose of the Heuristic Initialization step is to create a configuration file that specifies the signature (number and type of parameters) of the heuristic function. This information is required during the Heuristic Execution step.

Most importantly from a conceptual perspective, the compiler invokes Finch to execute the heuristic function. These steps are labeled Compilation and Heuristic Execution in the figure. Finally, Finch invokes an application-specific fitness evaluation to evaluate the result of the compilation (not shown).

2. Methods, Assumptions, and Procedures

The first phase of the program consisted of completing literature reviews related to the Raw and TRIPS architectures, obtaining existing copies of the compilers and simulators, and verifying their correct operation. The second phase consisted of developing parallel EC implementations for both architectures. This directly satisfied one of the objectives of the project. More importantly, it facilitated deeper understanding of the architectures, as well as the use of the compilers and the simulators. Given that understanding, the next phase consisted of more completely identifying and understanding the opportunities for optimization, the algorithms used by the existing compilers, and their implementations. In the fourth phase, we integrated an evolutionary algorithm with the in-lining optimization algorithm used by the existing TRIPS compiler. The final phase consisted of empirically evaluating programs generated using the enhanced compiler.

2.1 ISI's Raw Workstation

Information Science Institute East (ISI-East) maintains the Raw toolchain (e.g. compiler, simulator) and a Raw handheld board on a LINUX workstation (crudo.east.isi.edu). With assistance from Jinwoo Suh of ISI-East, the Raw toolchain was installed under the researcher's user accounts on crudo, and the correct installation was verified by the completion of the tutorials included with the Raw starsearch distribution, as well as the installation, compilation, and simulation of a suite of benchmarks provide by the Raw group.

Finally, several programs were developed from scratch that use various features of the Raw architecture, including an island model parallel EA. Collectively, these programs used both the static and dynamic communication features of the devices. These programs were simulated successfully using the Raw simulator and executed on the Raw board connected to crudo.

In the process of implementing these programs it was determined that the Raw toolchain places the parallel decomposition and mapping responsibilities on the programmer. As explained above, this limitation of the toolchain necessitated the focus of subsequent effort entirely on the TRIPS architecture.

2.2 RHIT's TRIPS Workstation

The a02 release of the TRIPS toolchain was installed on a Rose-Hulman Institute of Technology Linux workstation (clive.cs.rose-hulman.edu). A series of steps was taken to develop familiarity with the TRIPS toolchain, intermediate language, and assembly language:

- Correct installation of the software was verified by compiling and simulating the torture tests that ship with the TRIPS installer.
- The ability to perform edit-compile-simulation cycles was verified by compiling and simulating minor variations on the torture tests.
- The ability to develop TRIPS software from scratch was verified. Specifically, two different versions of a simple

evolutionary algorithm were compiled and simulated. The researchers were pleasantly surprised by the level of compatibility between the TRIPS compiler and gcc.

- The ability to develop and simulate both TRIPS Intermediate Language (TIL) and TRIPS Assembly Language (TASL) programs from scratch was verified, progressing from simple programs to programs that included branching, function calls, and text output.
- Comparisons were made between hand coded and compiler generated TRIPS assembly language (TASL), along with the TRIPS Intermediate Language (TIL) resulting in both cases. The comparison was performed using a pseudorandom number generator based on a 32-bit linear feedback shift register. The program was first hand-coded in TASL, and then an equivalent program was implemented using C. The TASL and TIL generated by the TRIPS toolchain (using various optimizations) was compared to the hand-coded versions.
- The ability to generate modified versions of the executables in the TRIPS toolchain was verified.

2.3 Classes of EC in Compilation Methods

Several classes of methods for the application of EC in enhancing compiler effectiveness have been identified. These methods are complementary, rather than mutually exclusive. They include:

- Compiler-algorithm-time. In this method, some variation of EC is used to evolve algorithms used within the compiler. This is the approach used by the Finch Meta Optimization tool. Rice University's technique of evolving the order of application of compiler optimizations also fits in this category [6]. This method has the advantage that the EA executes off-line (in the sense that it does not execute during the development of an application). Thus, compilation time will not necessarily increase. Also, the same source code will always be compiled to the same executable, which will always be scheduled in the same way. However, this method must be trained on some set of applications, and may result in a compiler that is less effective on applications outside of the training set.
- Compiler-parameterization-time. In this method, one or more EAs are used to evolve parameters of algorithms used within the existing compiler. An example of this approach is the Acovea tool, in which compiler optimization flags are chosen by an EA [8]. This method shares the advantages and disadvantages of the compiler-algorithm-time method. Relative to that method, this one has the advantage that the search spaces on which the EAs operate are more structured. The disadvantage is that the optimal parameterizations of existing algorithms may be less effective than yet-to-be identified algorithms.
- Compile-time. In this method, one or more EAs are included in the compiler. An advantage of this approach is that all of the details of the application are available to the EA, so there is some probability of finding the optimal machine instruction sequence and the optimal execution schedule. However, the same source code will produce different executables each time it is compiled, which would be a major disadvantage in a production environment.
- Schedule-time. This idea provided the original motivation for this effort. In this method, an EA is included in the scheduler. An advantage of this approach is that the execution time of candidate schedules can be modeled exactly, so the quality of the schedule is determined entirely by the effectiveness of the EA. However, the EA has no influence on the effectiveness of the compilation process. Perhaps more importantly, the same executable is scheduled differently each time it is executed, which may be a major disadvantage in an application development environment.

2.4 Integration of Finch and TRIPS Toolchain

With this groundwork laid, effort was then focused on the application of the Finch metaoptimization tool to the TRIPS toolchain, and specifically to the Scale compiler targeting the TRIPS architecture. Integration of Finch into any tool requires the integration of three library calls into the tool, corresponding to the three call backs illustrated in Figure 2. The directory structure of Scale is shown in Figure 3. In order to integrate Finch with Scale, the following library calls were implemented:

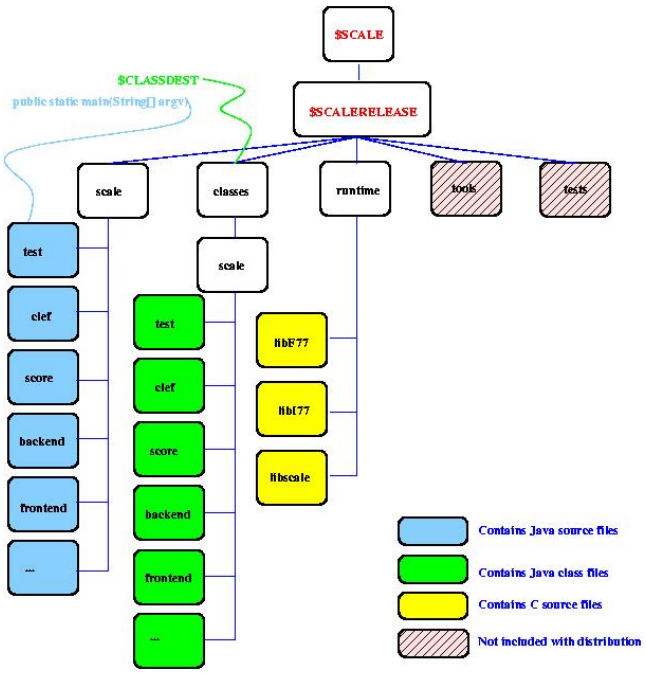

Figure 3. Scale Directory Structure (http://www-ali.cs.umass.edu/Scale)

- Immediately after parsing the command line arguments, the `compile()` method of `test/Scale.java` invokes `Finch.initializeLib()`. This call initializes Finch's evaluation of specific heuristic functions under consideration.
- The `getPriority()` method of `score/trans/Inlining.java` invokes

`Finch.evaluateReal(doubleArgs, boolArgs)`. This call causes Finch to evaluate candidate heuristic functions.

- Immediately before terminating the execution of Scale, the `compile()` method invokes `Finch.finalizeLib()`. This call finalizes Finch's evaluation of candidate heuristic functions.

The arguments passed to `Finch.evaluateReal()` are

- the largest allowable size of the executable
- the current size of the executable
- the level of purity[2] of the routine
- the size of the routine
- the number of AST children of this node (which is always 1)
- the number of function pointers in the body of the routine
- the number of routines that call this routine

An attempt was made to experimentally evaluate the effectiveness of Finch for optimizing the in-lining priority function in the TRIPS compiler, using an extremely small population size and generation count, and a modestly sized program as input to the compiler. However, this experiment did not complete given six CPU-days of execution time. It was thus concluded that multiprocessor environments would be required to obtain meaningful experimental results.

2.5 RHIT's TRIPS Clusters

The Finch metaoptimization tool was installed on a cluster of Windows 2000 workstations (in the Rose-Hulman Institute of Technology Department of Computer Science and Software Engineering OS/Security laboratory). This cluster was chosen because it provided the Network File System (NFS) and Portable Batch System (PBS) services, both of which are required by the downloadable version of Finch. Correct installation of Finch was verified using the provided test cases, after which the TRIPS toolchain was installed on the cluster and integrated with Finch as described above. A single computational experiment using this cluster produced an in-lining priority function that resulted in an 8% reduction in the static ratio of hyberblocks to instructions per hyberblock over the set of test programs employed.

The workstations that made up the OS/Security laboratory cluster are intended for student use and are re-imaged periodically as different courses make use of the laboratory. Therefore, it was necessary to port both the TRIPS toolchain and Finch to the Rose-Hulman Institute of Technology "Beowulf" Linux cluster (brain.rose-hulman.edu). However, the Beowulf cluster does not provide NFS and PBS, so it was also necessary to modify Finch's interprocessor communication to make use of the Message Passing Interface (MPI) standard, which is supported on the cluster.

[2] Scale has seven levels of "purity" associated with various combinations of the following characteristics: side effects, global variable references, and modification of memory locations referenced by arguments.

2.6 Metrics

Hardware execution of TRIPS applications was not an option in this effort, so the effectiveness of the techniques developed in this effort is evaluated on the basis of proxies for execution time of selected applications. Individual applications developed in the process of installing and verifying the TRIPS toolchain were evaluated on the basis of both detailed processor timing simulation (`tsim_proc`) and architectural simulation (`tsim_arch`). The former is cycle-accurate, while the latter reports the count of hyperblocks executed (as well as other statistics). Because the TRIPS architecture executes hyperblocks atomically, total execution time is closely related to the number of hyperblocks executed.

Neither the detailed processor timing simulation nor the architectural simulation is computationally efficient enough for repeated use as part of an EA's fitness evaluation function. As such, further approximations to execution time were made based on static evaluations of the executables produced by the EA. One approximation used was the count of hyperblocks in the executable, which is roughly proportional to the hyperblock execution count for the benchmarks used in this effort. Another was the average number of instructions per hyperblock, which is inversely related to hyperblock count for fixed instruction counts. This was chosen as the primary metric for this effort, based on observations by the TRIPS developers that the key to good performance is maximizing the number of instructions per hyperblock.

3. Results and Discussion

Several established benchmarks were considered, each of which contains multiple functions of varying size and invocation frequency. Each of the candidate benchmarks was compiled to TRIPS Intermediate Language (TIL) using a range of values for the allowable code bloat, and the fitness of the resulting TIL evaluated. Finch seeks to minimize the provided fitness function. Thus, for these experiments and the others discussed in this section, fitness was computed by subtracting the average number of instructions per hyperblock from the number of possible instructions per hyperblock (128).

One of the candidate benchmarks, the "Ground Moving Target Indicator (GMTI)," yielded a fitness that varied gradually with allowable code bloat (see Figure 4), and thus represents a test case against which candidate in-lining priority functions can be ranked. None of the SPEC_CPU2000v1.3 benchmarks exhibited this property.

A series of computational experiments applying Finch in the generation of in-lining heuristic functions for the Scale compiler and using the GMTI benchmark as a test case was executed on the RHIT Beowulf cluster using 17 processors. Experiments performed during the development of the software yielded fitness values equal to that produced by the unmodified compiler, which is assumed to be globally optimal. However, none of the experiments performed on the final version of the software did.

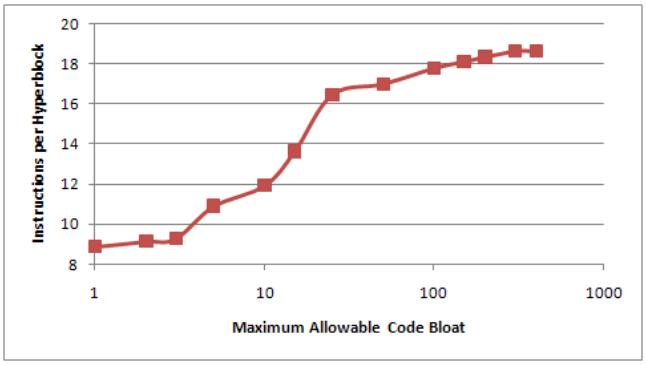

Figure 4. In-lining effectiveness as function of allowable code bloat for the unmodified Scale compiler and the GMTI benchmark.

For each generation of each experiment, the average number of instructions per hyperblock resulting from the use of each candidate priority function was calculated. The maximum, mean, and minimum of this value in a representative experiment are shown as a function of generation number in Figure 5.

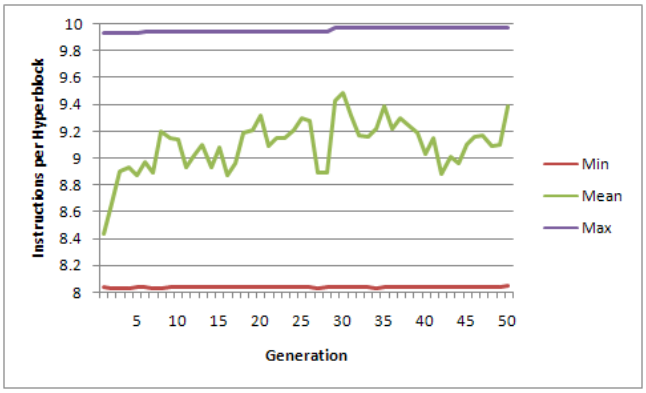

Figure 5. Representative results of Finch Meta-optimization of Scale in-lining, using the GMTI benchmark, a maximum allowable code bloat of 10%, a population size of 64, a maximum expression height of 4, and a mortality rate of 99%.

Several small improvements in the best heuristic function occur over the course of execution, resulting in an overall increase in average instructions per hyperblock of approximately 0.5%. The positive trend in the mean of the average instructions per hyperblock is easier to see, indicating that recombining features of good candidate priority functions tends to result in the construction of better candidate priority functions.

4. Conclusions

At the outset of this effort, the primary goal was to enhance the scheduling of instructions for both the Raw and TRIPS architectures by modifying their compilers to combine evolutionary computation (EC) techniques with the existing algorithms. Early on, it became apparent that this goal was not feasible with respect to the Raw architecture because the burden of program decomposition and mapping currently rests on the programmer. Thus, the remainder of the effort focused on the TRIPS architecture.

Four general techniques were identified for the application of EC in enhancing compiler effectiveness. The Finch Meta-optimization Framework implementation of the compiler-algorithm-time technique was adopted for this effort. Other techniques include compiler-parameterization-time, compile-time, and schedule-time. Tradeoff considerations among these techniques include their impact on execution time, compilation time, compiler construction time, reproducibility of execution, reproducibility of compilation, and breadth of application space targeted.

Given the advantages and disadvantages of each of the techniques, various combinations are appropriate in specific situations. One of the motivations for PCAs is to achieve near-optimal performance on each mission-critical application in a dynamic workload. For such an application, it is reasonable to assume that it is worthwhile to invest considerable offline computational effort in order to obtain improvements in online performance. As such, the compiler-algorithm time, compiler-parameterization time, and compile-time techniques should all be considered. Furthermore, assuming that the set of critical applications in the workload is small, the compiler-algorithm time and compiler-parameterization time techniques are especially applicable, since their application tailors the compiler to the applications in the training set, which can be chosen to consist of exactly the applications of interest. The schedule-time technique also has potential applicability in the context of PCAs, but execution time predictability must be addressed before it is practical for use in operational environments.

In order to integrate the Finch Metaoptimization Framework with the Scale compiler used in the TRIPS toolchain, several modifications to the compiler were implemented. Immediately after parsing its command line arguments, the modified version of Scale invokes a Finch method that prepares for the evaluation of a candidate priority function generated by the evolutionary algorithm. Also, the method that is normally used to compute the priority function that is built into Scale was modified to instead invoke a second Finch method that evaluates the candidate priority function. Finally, immediately before terminating, the modified version of Scale invokes a third Finch method to finalize the evaluation of the candidate priority function.

Computational experiments were performed to evaluate the effectiveness of Finch in evolving in-lining priority functions for Scale. The experiments were executed on the Rose-Hulman Institute of Technology Beowulf cluster. This required porting both the TRIPS toolchain and Finch to the cluster, as well as modifying Finch's interprocessor communication to make use of the Message Passing Interface (MPI) standard. The average number of instructions per generated hyperblock was used as the primary metric for these experiments, based in part on observations by the TRIPS developers that maximizing this metric is essential to achieving good performance. A number of applications were considered as possible inputs to Scale for the experiments. The "Ground Moving Target Indicator (GMTI)," was chosen because for the unmodified version of Scale the chosen metric varies gradually with the allowable code bloat parameter. Using small population sizes and small generation counts, the software occasionally obtains values of the metric equal to that produced by the unmodified compiler, but not reliably. Each experiment requires between three and four hours of wall clock time using 17 processors.

5. Recommendations

The limited success of the computational experiments described in this report should be interpreted in light of the fact that by genetic programming standards, the population size and generation count for these experiments are both extremely small. It is likely that larger values of either parameter would result in the identification of more effective in-lining priority functions. Furthermore, each experiment required less than four hours to execute, so using larger population sizes and generation counts would not result in prohibitive execution times.

This effort has laid the groundwork for the development of hybrid evolutionary algorithms that exploit both the global search properties of evolutionary computation and the effectiveness of the existing compiler optimization algorithms. Future research is needed in a number of areas:

- Perform additional computational experiments related to TRIPS in-lining, as well as similar experiments for other compiler optimizations involving priority functions (e.g. loop unrolling). These experiments can be completed without further modification of Finch. The advantage of those kinds of optimizations is that they have relatively direct impact on the formation of hyperblocks (which is where the greatest impact on performance can be made). The limitation is that they explore relatively small parts of the space of assembly language programs.
- Explore larger areas of the space of TRIPS assembly language programs by modifying Scale so that a Finch-optimized priority function controls the building of hyperblocks. This could be done in a few different ways. The most promising of these is modifying the control flow graph (CFG) creation function so that it consults the finch-optimized priority function. This would allow Finch to change the CFG so that it will make "better" hyperblocks, since the fitness function uses a heuristic that only takes into account the average number of instructions per hyperblock.
- The spatial distribution of an EA across a tile-based architecture to provide dynamic performance optimization still merits investigation.

6. ACKNOWLEDGMENTS

This work was supported by the Advanced Computing Architectures Branch, Information Directorate, Air Force Research Laboratory, Air Force Materiel Command, USAF, under grant FA8750-05-1-0019. The Principal Investigator was assisted in this effort by Tyler Hicks-Wright, Matt Ellis, and Mike McClurg. The team is indebted to several other individuals for their help in establishing the infrastructure necessary to perform the work. Jinwoo Suh of Information Sciences Institute East provided user accounts on ISI-East's Raw workstation, as well as guidance and assistance in installing and using the Raw toolchain and using the Raw handheld board. Doug Burger and Steve Kecklin of University of Texas Austin provided access to the TRIPS toolchain and made themselves available for discussions about the compiler's optimization algorithms at several meetings of the Polymorphous Computing Architectures Principal Investigators. Numerous other individuals within both organizations answered questions and provided assistance at various times during the effort.

7. REFERENCES

[1] Agarwal, A. (1999, August). Raw Computation. *Scientific American* .

[2] Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[3] Burger, D., Keckler, S., McKinley, K., Dahlin, M., John, L., Lin, C., et al. (2004, July). Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer* , pp. 44-55.

[4] Burger, D., Keckler, S., McKinley, K., Lin, C., Dahlin, M., Nowka, K., et al. (2004). *TRIPS: Tera-op Reliable Intelligently adaptive Processing System.* Monterey, CA: DARPA Polymorphous Computing Architectures Program PI Meeting.

[5] Cantú-Paz, E. (1998). *A Survey of Parallel Genetic Algorithms.* http://citeseer.ist.psu.edu/155991.html.

[6] Cooper, K., Grosul, A., Harvey, T., Reeves, S., Subramanian, D., Torczon, L., et al. (2004). Exploring the Structure of the Space of Compilation Sequences Using Randominzed Search Algorithms. *2004 LACSI Symposium.* Santa Fe, NM.

[7] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

[8] Ladd, S. R. (1996). *Acovea Overview*. Retrieved May 14, 2007, from Coyote Gulch Productions: http://www.coyotegulch.com/products/acovea/

[9] Lebak, J. (2004). *Application Analysis, Kernel Benchmarks, and PCA Testbed Update.* Monterey, CA: DARPA Polymorphous Computing Architectures Program PI Meeting.

[10] Rabbah, R. M., Agarwal, A., & Amarasinghe, S. (2004). *Update on Raw and StreamIt.* Monterey, CA: DARPA Polymorphous Computing Architectures Program PI Meeting.

[11] Scale Compiler Group. (n.d.). Retrieved May 14, 2007, from Scale Home Page: http://www-ali.cs.umass.edu/Scale

[12] Stephenson, M., Martin, M., O'Reilly, U., & Amarasinghe, S. (2003). Meta Optimization: Improving Compiler Heuristics with Machine Learning. *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation.* San Diego, CA.

[13] Weaver, G. E., Cahoon, B. D., Moss, J. E., McKinley, K. S., Wright, E. J., & Burrill, J. H. (1997). *The Common Language Encoding Form (CLEF) Design Document.* Amherst, MA: University of Massachusetts at Amherst.