Meta Optimization: Improving Compiler Heuristics with Machine Learning

Mark Stephenson and Saman Amarasinghe
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139
{mstephen, saman}@cag.lcs.mit.edu

Martin Martin and Una-May O'Reilly
Massachusetts Institute of Technology
Artificial Intelligence Laboratory
Cambridge, MA 02139
{mcm, unamay}@ai.mit.edu

ABSTRACT

Compiler writers have crafted many heuristics over the years to approximately solve NP-hard problems efficiently. Finding a heuristic that performs well on a broad range of applications is a tedious and difficult process. This paper introduces Meta Optimization, a methodology for automatically fine-tuning compiler heuristics. Meta Optimization uses machine-learning techniques to automatically search the space of compiler heuristics. Our techniques reduce compiler design complexity by relieving compiler writers of the tedium of heuristic tuning. Our machine-learning system uses an evolutionary algorithm to automatically find effective compiler heuristics. We present promising experimental results. In one mode of operation Meta Optimization creates application-specific heuristics which often result in impressive speedups. For hyperblock formation, one optimization we present in this paper, we obtain an average speedup of 23% (up to 73%) for the applications in our suite. Furthermore, by evolving a compiler's heuristic over several benchmarks, we can create effective, general-purpose heuristics. The best general-purpose heuristic our system found for hyperblock formation improved performance by an average of 25% on our training set, and 9% on a completely unrelated test set. We demonstrate the efficacy of our techniques on three different optimizations in this paper: hyperblock formation, register allocation, and data prefetching.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques; I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Design, Algorithms, Performance

Keywords

Machine Learning, Priority Functions, Genetic Programming, Compiler Heuristics

PLDI'03, June 9–11, 2003, San Diego, California, USA.

1. INTRODUCTION

Compiler writers have a difficult task. They are expected to create effective and inexpensive solutions to NP-hard problems such as register allocation and instruction scheduling. Their solutions are expected to interact well with other optimizations that the compiler performs. Because some optimizations have competing and conflicting goals, adverse interactions are inevitable. Getting all of the compiler passes to mesh nicely is a daunting task.

The advent of intractably complex computer architectures also complicates the compiler writer's task. Since it is impossible to create a simple model that captures the intricacies of modern architectures and compilers, compiler writers rely on inaccurate abstractions. Such models are based upon many assumptions, and thus may not even properly simulate first-order effects.

Because compilers cannot afford to optimally solve NP-hard problems, compiler writers devise clever heuristics that quickly find good approximate solutions for a large class of applications. Unfortunately, heuristics rely on a fair amount of tweaking to achieve suitable performance. Trial-and-error experimentation can help an engineer optimize the heuristic for a given compiler and architecture. For instance, one might be able to use iterative experimentation to figure out how much to unroll loops for a given architecture (*i.e.*, without thrashing the instruction cache or incurring too much register pressure).

After studying several compiler optimizations, we found that many heuristics have a focal point. A single *priority* or *cost* function often dictates the efficacy of a heuristic. A priority function— a function of the factors that affect a given problem— measures the relative importance of the different options available to a compiler algorithm.

Take register allocation for example. When a graph coloring register allocator cannot successfully color an interference graph, it spills a variable to memory and removes it from the graph. The allocator then attempts to color the reduced graph. When a graph is not colorable, choosing an appropriate variable to spill is crucial. For many allocators, this decision is bestowed upon a single priority function. Based on relevant data (*e.g.*, number of references, depth in loop nest, etc.), the function assigns weights to all uncolored variables and thereby determines which variable to spill.

Fine-tuning priority functions to achieve suitable performance is a tedious process. Currently, compiler writers manually experiment with different priority functions. For instance, Bernstein et al. manually identified three priority functions for choosing spill variables [3]. By applying the three functions to a suite of benchmarks, they found that a register allocator's effectiveness is highly dependent on the priority function the compiler uses.

The importance of priority functions is a key insight that motivates Meta Optimization, a method by which a machine-learning algorithm automatically searches the priority function solution space. More specifically, we use a learning algorithm that iteratively searches for priority functions that improve the execution time of compiled applications.

Our system can be used to cater a priority function to a specific input program. This mode of operation is essentially an advanced form of feedback directed optimization. More importantly, it can be used to find a general-purpose function that works well for a broad range of applications. In this mode of operation, Meta Optimization can perform the tedious work that is currently performed by engineers. For each of the three case studies we describe in this paper, we were able to at least match the performance of human-generated priority functions. In some cases we achieved considerable speedups.

While many researchers have used machine-learning techniques and exhaustive search algorithms to improve an application, none have used learning to search for priority functions. Because Meta Optimization improves the effectiveness of the compiler itself, in theory, we need only apply the process once (rather than on a per-application basis).

The remainder of this paper is organized as follows. The next section introduces priority functions. Section 3 describes genetic programming, a machine-learning technique that is well suited to our problem. Section 4 discusses our methodology. We apply our technique to three separate case studies in Section 5, Section 6, and Section 7. Results of our experiments are included in the case study sections. Section 8 discusses related work, and finally Section 9 concludes.

2. PRIORITY FUNCTIONS

This section is intended to give the reader a feel for the utility and ubiquity of priority functions. Put simply, priority functions prioritize the options available to a compiler algorithm.

For example, in list scheduling, a priority function assigns a weight to each instruction in the scheduler's dependence graph, dictating the order in which to schedule instructions. A common and effective heuristic assigns priorities using latency-weighted depths [10]. Essentially, this is the instruction's depth in the dependence graph, taking into account the latency of instructions on all paths to the root nodes:

$$P(i) = \begin{cases} latency(i) & : \text{ if } i \text{ is independent.} \\ \max\limits_{\text{i depends on j}} latency(i) + P(j) & : \text{ otherwise.} \end{cases}$$

The list scheduler proceeds by scheduling *ready* instructions in priority order. In other words, if two instructions are ready to be scheduled, the algorithm will favor the instruction with the higher priority. The scheduling algorithm hinges upon the priority function. Apart from enforcing the legality of the schedule, the scheduler entirely relies on the priority function to make all of its decisions.

This description of list scheduling is a simplification. Production compilers use sophisticated priority functions that account for many competing factors (*e.g.*, how a given schedule may affect register allocation).

The remainder of the section lists a few other priority functions that are amenable to the techniques we discuss in this paper. We will explore three of the following priority functions in detail later in the paper.

- **Clustered scheduling**: Özer et al. describe an approach to scheduling for architectures with clustered register files [20]. They note that the choice of priority function has a "strong effect on the schedule." They also investigate five different priority functions [20].
- **Hyperblock formation**: Later in this paper we use the formation of predicated hyperblocks as a case study.
- **Meld scheduling**: Abraham et al. rely on a priority function to schedule across region boundaries [1]. The priority function is used to sort regions by the order in which they should be visited.
- **Modulo scheduling**: In [22], Rau states that "there is a limitless number of priority functions" that can be devised for modulo scheduling. Rau describes the tradeoffs involved when considering scheduling priorities.
- **Data Prefetching**: Later in this paper we investigate a priority function that determines whether or not to prefetch an address.
- **Register allocation**: Many register allocation algorithms use cost functions to determine which variables to spill if spilling is required. We use register allocation as a case study later in the paper.

This is not an exhaustive list of applications. Many important compiler optimizations employ cost functions of the sort mentioned above. The next section introduces genetic programming, which we use to automatically find effective priority functions.

3. GENETIC PROGRAMMING

Of the many available machine-learning techniques, we chose to employ genetic programming (GP) because its attributes best fit the needs of our application. The following list highlights the suitability of GP to our problem:

- GP is especially appropriate when the relationships among relevant variables are poorly understood [13]. Such is the case with compiler heuristics, which often feature uncertain tradeoffs. Today's complex systems also introduce uncertainty.
- GP is capable of searching high-dimensional spaces. Many other learning algorithms are not as scalable.
- GP is a distributed algorithm. With the cost of computing power at an all-time low, it is now economically feasible to dedicate a cluster of machines to searching a solution space.
- GP solutions are human readable. The 'genomes' on which GP operates are parse trees which can easily be converted to free-form arithmetic equations. Other machine-learning representations, such as neural networks, are not as comprehensible.

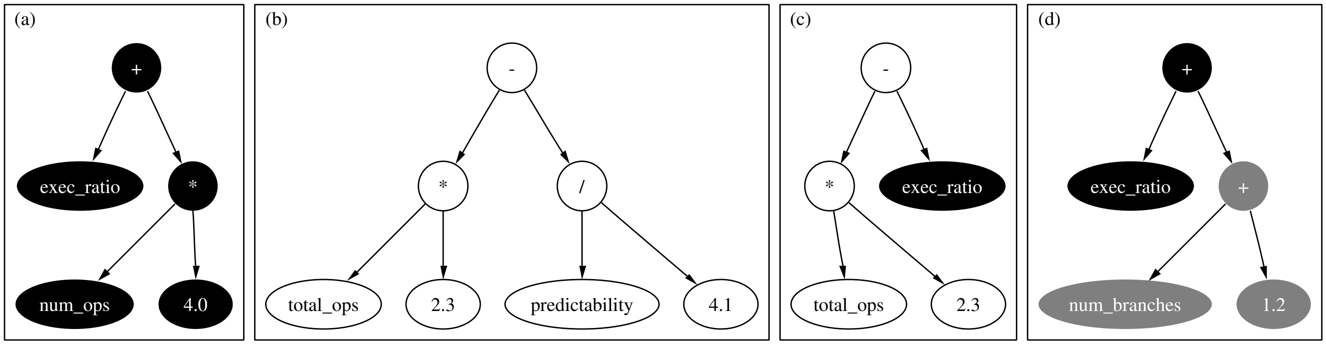

Figure 1: GP Genomes. Part (a) and (b) show examples of GP genomes. Part (c) provides an example of a random crossover of the genomes in (a) and (b). Part (d) shows a mutation of the genome in part (a).

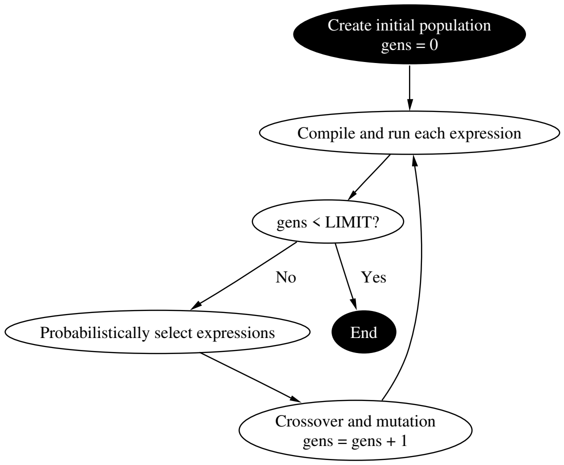

Figure 2: Flow of genetic programming. Genetic programming (GP) initially creates a population of expressions. Each expression is then assigned a fitness, which is a measure of how well it satisfies the end goal. In our case, fitness is proportional to the execution time of the compiled application(s). Until some user-defined cap on the number of generations is reached, the algorithm probabilistically chooses the best expressions for mating and continues. To guard against stagnation, some expressions undergo mutation.

Like other evolutionary algorithms, GP is loosely patterned on Darwinian evolution. GP maintains a population of parse trees [13]. In our case, each parse tree is an expression that represents a priority function. As with natural selection, expressions are chosen for reproduction (called crossover) according to their level of fitness. Expressions that best solve the problem are most likely to have progeny. The algorithm also randomly mutates some expressions to innovate a possibly stagnant population.

Figure 2 shows the general flow of genetic programming in the context of our system. The algorithm begins by creating a population of initial expressions. The baseline heuristic over which we try to improve is included in the initial population; the remainder of the initial expressions are randomly generated. The algorithm then determines each expression's level of fitness. In our case, compilers that produce the *fastest* code are fittest. Once the algorithm reaches a user-defined limit on the number of generations, the process stops; otherwise, the algorithm proceeds by probabilistically choosing the best expressions for mating. Some of the offspring undergo mutation, and the algorithm continues.

Unlike other evolutionary algorithms, which use fixed-length binary *genomes*, GP's expressions are variable in length and free-form. Figure 1 provides several examples of genetic programming genomes (expressions). Variable-length genomes do not artificially constrain evolution by setting a maximum genome size. However, without special consideration, genomes grow exponentially during crossover and mutation.

Our system rewards *parsimony* by selecting the smaller of two otherwise equally fit expressions [13]. Parsimonious expressions are aligned with our philosophy of using GP as a tool for compiler writers and architects to identify important heuristic *features* and the relationships among them. Without enforcing parsimony, expressions quickly become unintelligible.

In Figure 1, part (c) provides an example of crossover, the method by which two expressions reproduce. Here the two expressions in (a) and (b) produce offspring. Crossover works by selecting a random node in each parent, and then swapping the subtrees rooted at those nodes[1]. In theory, crossover works by propagating 'good' subexpressions. Good subexpressions increase an expression's fitness.

Because GP favors fit expressions, expressions with favorable building blocks are more likely selected for crossover, further disseminating the blocks. Our system uses tournament selection to choose expressions for crossover. Tournament selection chooses N expressions at random from the population and selects the one with the highest fitness [13]. N is referred to as the tournament size. Small values of N reduce selection pressure; expressions are only compared against the other $N-1$ expressions in the tournament.

Finally, part (d) shows a mutated version of the expression in (a). Here, a randomly generated expression supplants a randomly chosen node in the expression. For details on the mutation operators we implemented, see [2].

[1] Selection algorithms must use caution when selecting random tree nodes. If we consider a full binary tree, then leaf nodes comprise over 50% of the tree. Thus, a naive selection algorithm will choose leaf nodes over half of the time. We employ depth-fair crossover, which equally weighs each level of the tree [12].

Real-Valued Function	Representation
$Real_1 + Real_2$	(add $Real_1$ $Real_2$)
$Real_1 - Real_2$	(sub $Real_1$ $Real_2$)
$Real_1 \cdot Real_2$	(mul $Real_1$ $Real_2$)
$\begin{cases} Real_1/Real_2 & : \text{ if } Real_2 \neq 0 \\ 0 & : \text{ if } Real_2 = 0 \end{cases}$	(div $Real_1$ $Real_2$)
$\sqrt{Real_1}$	(sqrt $Real_1$)
$\begin{cases} Real_1 & : \text{ if} Bool_1 \\ Real_2 & : \text{ if not} Bool_1 \end{cases}$	(tern $Bool_1$ $Real_1$ $Real_2$)
$\begin{cases} Real_1 \cdot Real_2 & : \text{ if} Bool_1 \\ Real_2 & : \text{ if not} Bool_1 \end{cases}$	(cmul $Bool_1$ $Real_1$ $Real_2$)
Returns real constant K	(rconst K)
Returns real value of arg from environment	(rarg arg)

Boolean-Valued Function	Representation
$Bool_1$ and $Bool_2$	(and $Bool_1$ $Bool_2$)
$Bool_1$ or $Bool_2$	(or $Bool_1$ $Bool_2$)
not $Bool_1$	(not $Bool_1$)
$Real_1 < Real_2$	(lt $Real_1$ $Real_2$)
$Real_1 > Real_2$	(gt $Real_1$ $Real_2$)
$Real_1 = Real_2$	(eq $Real_1$ $Real_2$)
Returns Boolean constant	(bconst $\{true, false\}$)
Returns Boolean value of arg from environment	(barg arg)

Table 1: GP primitives. Our GP system uses the primitives and syntax shown in this table. The top segment represents the real-valued functions, which all return a real value. Likewise, the functions in the bottom segment all return a Boolean value.

To find general-purpose expressions (*i.e.*, expressions that work well for a broad range of input programs), the learning algorithm learns from a *set* of 'training' programs. To train on multiple input programs, we use the technique described by Gathercole in [9]. The technique—called dynamic subset selection (DSS)— trains on subsets of the training programs, concentrating more effort on programs that perform poorly compared to the baseline heuristics. DSS reduces the number of fitness evaluations that need to be performed in order to achieve a suitable solution. Because our system must compile and run benchmarks to test an expression's level of fitness, fitness evaluations for our problem are costly.

The next section describes the methodology that we use throughout the remainder of the paper.

4. METHODOLOGY

Compiler priority functions are often based on assumptions that may not be valid across application and architectural variations. In other words, who knows on what set of benchmarks, and for what target architecture the priority functions were designed? It could be the case that a priority function was designed for completely orthogonal circumstances than those under which you use your compiler.

Our system uses genetic programming to automatically search for effective priority functions. Though it may be possible to 'evolve' the underlying algorithm, we restrict ourselves to priority functions. This drastically reduces search space size, and the underlying algorithm ensures optimization legality. Furthermore, this technique is still very powerful; even small changes to the priority function can drastically improve (or diminish) performance.

We optimize a given priority function by wrapping the iterative framework of Figure 2 around the compiler and architecture. We replace the priority function that we wish to optimize with an expression parser and evaluator. This allows us to compile the benchmarks in our 'training' suite using the expressions— which are priority functions— in the population. The expressions that create the fastest executables for the applications in the training suite are favored for crossover.

Parameter	Setting
Population size	400 expressions
Number of generations	50 generations
Generational replacement	22 expressions
Mutation rate	5%
Tournament size	7
Elitism	Best expression is guaranteed survival.
Fitness	Average speedup over the baseline on the suite of benchmarks.

Table 2: GP parameters. This table shows the GP parameters we used to collect the results in this section.

Our system uses total execution time to assign fitnesses. This approach focuses on frequently executed procedures, and therefore, may slowly converge upon general-purpose solutions. However, when one wants to specialize a compiler for a given input program, this evaluation of fitness works extremely well.

Table 1 shows the GP expression primitives that our system uses. Careful selection of GP primitives is essential. We want to give the system enough flexibility to potentially find unexpected results. However, the more leeway we give GP, the longer it will take to converge upon a general solution.

Our system creates an initial population that consists of 399 randomly generated expressions; it randomly 'grows' expressions of varying heights using the primitives in Table 1 and *features* extracted by the compiler writer. Features are measurable program characteristics that the compiler writer thinks may be important for forming good priority functions (*e.g.*, latency-weighted depth for list scheduling).

In addition to the randomly generated expressions, we seed the initial population with the compiler writer's best guess. In other words, we include the priority function distributed with the compiler. For two of the three optimizations presented in this paper, we found that the seed is quickly obscured and weeded out of the population as more favorable expressions emerge. In fact, for hyperblock selection and data prefetching, which we discuss later, the seed had no impact on the final solution. These results suggest that one could use Meta Optimization to construct priority functions from scratch rather than trying to improve upon preexisting functions. In this way, our tool can reduce the complexity of compiler design by sparing the engineer from perfunctory algorithm tweaking.

Table 2 summarizes the parameters that we use to collect results. We chose the parameters in the table after a moderate amount of experimentation. We give our GP system 50 generations to find a solution. For the benchmarks that we surveyed, the time required to run for 50 generations is about one day per benchmark in the training set[2]. Our system memoizes benchmark fitnesses because fitness evaluations are so costly.

After every generation the system *randomly* replaces 22% of the population with new expressions created via the crossover

[2] We ran on 15 to 20 machines in parallel for the experiments in Section 5 and Section 6, and we used 5 machines for the experiments in Section 7.

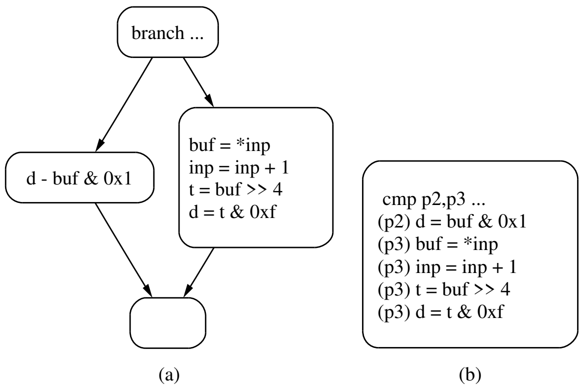

Figure 3: Control flow *v.* predicated execution. Part (a) shows a segment of control-flow that demonstrates a simple if-then-else statement. As is typical with multimedia and integer applications, there are few instructions per basic block in the example. Part (b) is the corresponding predicated hyperblock. If-conversion merges disjoint paths of control by creating predicated hyperblocks. Choosing which paths to merge is a balancing act. In this example, branching may be more efficient than predicating if p3 is rarely true.

operation presented in Section 3. Only the best expression is guaranteed survival. Typically, GP practitioners use much higher replacement rates. However, since we use dynamic subset selection, only a subset of benchmarks is evaluated in a generation. Thus, we need a lower replacement rate in order to increase the likelihood that a given expression will be tested on more than one subset of benchmarks. The mutation operator, which is discussed in the same section, mutates roughly 5% of the new expressions. Finally, we use a tournament size of 7 when selecting the fittest expressions. This setting causes moderate selection pressure.

The following three sections build upon the methodology described in this section by presenting individual case studies. Results for each of the case studies are included in their respective sections.

5. CASE STUDY I: HYPERBLOCK FORMATION

This section describes the operation of our system in the context of a specific compiler optimization: hyperblock formation. Here we introduce the optimization, and then we discuss factors that might be important when creating a priority function for it. We conclude the section by presenting experimental results for hyperblock formation.

Architects have proposed two noteworthy methods for decreasing the costs associated with control transfers[3]: improved branch prediction, and predication. Improved branch prediction algorithms would obviously increase processor utilization. Unfortunately, some branches are inherently unpredictable, and hence, even the most sophisticated algorithm would fail. For such branches, predication may be a fruitful alternative.

[3] The Pentium® 4 architecture features 20 pipeline stages. It squashes up to 126 in-flight instructions when it mispredicts.

Rather than relying on branch prediction, predication allows a multiple-issue processor to simultaneously execute the taken and fall-through paths of control flow. The processor nullifies all instructions in the incorrect path. In this model, a predicate operand guards the execution of every instruction. If the value of the operand is true, then the instruction executes normally. If however, the operand is false, the processor nullifies the instruction, preventing it from modifying processor state.

Figure 3 highlights the difference between control-flow and predicated execution. Part (a) shows a segment of control-flow. Using a process dubbed if-conversion, the IMPACT predicating compiler merges disjoint paths of execution into a predicated hyperblock. A hyperblock is a predicated single-entry, multiple-exit region. Part (b) shows the hyperblock corresponding to the control-flow in part (a). Here, `p2` and `p3` are mutually exclusive predicates that are set according to the branch condition in part (a).

Though predication effectively exposes ILP, simply predicating everything will diminish performance by saturating machine resources with useless instructions. However, an appropriate balance of predication and branching can drastically improve performance.

5.1 Feature Extraction

In the following list we give a brief overview of several criteria that are useful to consider when forming hyperblocks. Such criteria are often referred to as features. In the list, a path refers to a path of control flow (*i.e.*, a sequence of basic blocks that are connected by edges in the control flow graph):

- **Path predictability**: Predictable branches incur no misprediction penalties, and thus, should probably remain unpredicated. Combining multiple paths of execution into a single predicated region uses precious machine resources [15]. In this case, using machine resources to parallelize individual paths is typically wiser.

- **Path frequency**: Infrequently executed paths are probably not worth predicating. Including the path in a hyperblock would consume resources, and could negatively affect performance.

- **Path ILP**: If a path's level of parallelism is low, it may be worthwhile to predicate the path. In other words, if a path does not fully use machine resources, combining it with another sequential path probably will not diminish performance. Because predicated instructions do not need to know the value of their guarding predicate until late in the pipeline, a processor can sustain high levels of ILP.

- **Number of instructions in path**: Long paths use up machine resources, and if predicated, will likely slow execution. This is especially true when long paths are combined with short paths. Since every instruction in a hyperblock executes, long paths effectively delay the time to completion of short paths. The cost of misprediction is relatively high for short paths. If the processor mispredicts on a short path, the processor has to nullify all the instructions in the path, *and* the subsequent control-independent instructions fetched before the branch condition resolves.

- **Number of branches in path**: Paths of control through several branches have a greater chance of mispredicting. Therefore, it may be worthwhile to predicate such paths. On the other hand, including several such paths may produce large hyperblocks that saturate resources.

- **Compiler optimization considerations**: Paths that contain hazard conditions (*i.e.*, pointer dereferences and procedure calls) limit the effectiveness of many compiler optimizations. In the presence of hazards, a compiler must make conservative assumptions. The code in Figure 3(a) could benefit from predication. Without architectural support, the load from `*inp` cannot be hoisted above the branch. The program will behave unexpectedly if the load is not supposed to execute and it accesses protected memory. By removing branches from the instruction stream, predication affords the scheduler freer code motion opportunities. For instance, the predicated hyperblock in Figure 3(b) allows the scheduler to rearrange memory operations without control-flow concerns.

- **Machine-specific considerations**: A heuristic should account for machine characteristics. For instance, the branch delay penalty is a decisive factor.

Clearly, there is much to consider when designing a heuristic for hyperblock selection. Many of the above considerations make sense on their own, but when they are put together, contradictions arise. Finding the right mix of criteria to construct an effective priority function is nontrivial. That is why we believe automating the decision process is crucial.

5.2 Trimaran's Heuristic

We now discuss the heuristic employed by Trimaran's IMPACT compiler for creating predicated hyperblocks [15, 16]. The IMPACT compiler begins by transforming the code so that it is more amenable to hyperblock formation [15]. IMPACT's algorithm then identifies acyclic paths of control that are suitable for hyperblock inclusion. Park and Schlansker detail this portion of the algorithm in [21]. A priority function— which is the critical calculation in the predication decision process— assigns a value to each of the paths based on characteristics such as the ones just described [15]. Some of these characteristics come from runtime profiling.

IMPACT uses the priority function shown below:

$$h_i = \begin{cases} 0.25 & : \text{ if } path_i \text{ contains a hazard.} \\ 1 & : \text{ if } path_i \text{ is hazard free.} \end{cases}$$

$$d\_ratio_i = \frac{dep\_height_i}{\max_{j=1\rightarrow N} dep\_height_j}$$

$$o\_ratio_i = \frac{num\_ops_i}{\max_{j=1\rightarrow N} num\_ops_j}$$

$$priority_i = exec\_ratio_i \cdot h_i \cdot (2.1 - d\_ratio_i - o\_ratio_i) \quad (1)$$

The heuristic applies the above equation to all paths in a predicatable region. Based on a runtime profile, *exec_ratio* is the probability that the path is executed. The priority function also penalizes paths that contain hazards (*e.g.*, pointer dereferences and procedure calls). Such paths may constrain aggressive compiler optimizations. To avoid large hyperblocks, the heuristic is careful not to choose paths that have a large dependence height (*dep_height*) with respect to the maximum dependence height. Similarly it penalizes paths that contain too many instructions (*num_ops*).

IMPACT's algorithm then merges the paths with the highest priorities into a predicated hyperblock. The algorithm stops merging paths when it has consumed the target architecture's estimated resources.

Feature	Description
Registers	64 general-purpose registers, 64 floating-point registers, and 256 predicate registers.
Integer units	4 fully-pipelined units with 1-cycle latencies, except for multiply instructions, which require 3 cycles, and divide instructions, which require 8.
Floating-point units	2 fully-pipelined units with 3-cycle latencies, except for divide instructions, which require 8 cycles.
Memory units	2 memory units. L1 cache accesses take 2 cycles, L2 accesses take 7 cycles, and L3 accesses require 35 cycles. Stores are buffered, and thus require 1 cycle.
Branch unit	1 branch unit.
Branch prediction	2-bit branch predictor with a 5-cycle branch misprediction penalty.

Table 3: Architectural characteristics. This table describes the EPIC architecture over which we evolved. This model approximates the Intel Itanium architecture.

5.3 Experimental Setup

This section discusses the experimental results for optimizing Trimaran's hyperblock selection priority function. Trimaran is an integrated compiler and simulator for a parameterized EPIC architecture. Table 3 details the specific architecture over which we evolved. This model resembles Intel's Itanium® architecture.

We modified Trimaran's IMPACT compiler by replacing its hyperblock formation priority function (Equation 1) with our GP expression parser and evaluator. This allows IMPACT to read an expression and evaluate it based on the values of human-selected features that might be important for creating effective priority functions. Table 4 describes these features.

The hyperblock formation algorithm passes the features in the table as parameters to the expression evaluator. For instance, if an expression contains a reference to `dep_height`, the path's dependence height will be used when the expression is evaluated. Most of the characteristics in Table 4 were already available in IMPACT. Equation 1 has a local scope. To provide some global information, we also extract the minimum, maximum, mean, and standard deviation of all path-specific characteristics in the table.

We added a 2-bit dynamic branch predictor to the simulator and we modified the compiler's profiler to extract branch predictability statistics. Lastly, we enabled the following compiler optimizations: function inlining, loop unrolling, backedge coalescing, acyclic global scheduling [6], modulo scheduling [25], hyperblock formation, register allocation, machine-specific peephole optimization, and several classic optimizations.

Feature	Description
dep_height	The maximum instruction dependence height over all instructions in path.
num_ops	The total number of instructions in the path.
exec_ratio	How frequently this path is executed compared to other paths considered (from profile).
num_branches	The total number of branches in the path.
predictability	Average path predictability obtained by simulating a branch predictor (from profile).
predict_product	Product of branch predictabilities in the path (from profile).
avg_ops_executed	The average number of instructions executed in the path (from profile).
unsafe_JSR	If the path contains a subroutine call that may have side-effects, it returns *true*; otherwise it returns *false*.
safe_JSR	If the path contains a side-effect free subroutine call, it returns *true*; otherwise it returns *false*.
mem_hazard	If the path contains an unresolvable memory access, it returns *true*; otherwise it returns *false*.
max_dep_height	The maximum dependence height over all paths considered for hyperblock inclusion.
total_ops	The sum of all instructions in paths considered for hyperblock inclusion.
num_paths	Number of paths considered for hyperblock inclusion.

Table 4: Hyperblock selection features. The compiler writer chooses interesting attributes, and the system evolves a priority function based on them. We rely on profile information to extract some of these parameters. We also include the min, mean, max, and standard deviation of path characteristics. This provides some global information to the greedy local heuristic.

5.4 Experimental Results

We use the familiar benchmarks in Table 5 to test our system. All of the Trimaran certified benchmarks are included in the table[4] [24]. Our suite also includes many of the Mediabench benchmarks [14]. The build process for ghostscript proved too difficult to compile. We also exclude the remainder of the Mediabench applications because the Trimaran system does not compile them correctly[5].

We begin by presenting results for application-specialized heuristics. Following this, we show that it is possible to use Meta Optimization to create general-purpose heuristics.

5.4.1 Specialized Priority Functions

Specialized heuristics are created by optimizing a priority function for a given application. In other words, we train the priority function on a single benchmark. Figure 4 shows that Meta Optimization is extremely effective on a per-benchmark basis. The dark bar shows the speedup (over Trimaran's baseline heuristic) of each benchmark when run with the same data on which it was trained. The light bar shows the speedup attained when the benchmark processes a data set that was not used to train the priority function. We call this the novel data set.

[4] Due to preexisting bugs in Trimaran, we could not get 134.perl to execute correctly, though [24] certified it.

[5] We exclude cjpeg, the complement of djpeg, because it does not execute properly when compiled with some priority functions. Our system can also be used to uncover bugs!

Benchmark	Suite	Description
codrle4 decodrle4	See [4]	RLE type 4 encoder/decoder.
huff_enc huff_dec	See [4]	A Huffman encoder/decoder.
djpeg	Mediabench	Lossy still image decompressor.
g721encode g721decode	Mediabench	CCITT voice compressor/decompressor.
mpeg2dec	Mediabench	Lossy video decompressor.
rasta	Mediabench	Speech recognition application.
rawcaudio rawdaudio	Mediabench	Adaptive differential pulse code modulation audio encoder/decoder.
toast	Mediabench	Speech transcoder.
unepic	Mediabench	Experimental image decompressor.
085.cc1	SPEC92	gcc C compiler.
052.alvinn	SPEC92	Single-precision neural network training.
179.art	SPEC2000	A neural network-based image recognition algorithm.
osdemo mipmap	Mediabench Mediabench	Part of a 3-D graphics library similar to OpenGL.
129.compress	SPEC95	In-memory file compressor and decompressor.
023.eqntott	SPEC92	Creates a truth table from a logical representation of a Boolean equation.
132.ijpeg	SPEC95	JPEG compressor and decompressor.
130.li	SPEC95	Lisp interpreter.
124.m88ksim	SPEC95	Processor simulator.
147.vortex	SPEC95	An object oriented database.

Table 5: Benchmarks used. The set includes applications from the SpecInt, SpecFP, and Mediabench benchmark suites, as well as a few miscellaneous programs.

Intuitively, in most cases the training input data achieves a better speedup. Because Meta Optimization is performance-driven, it selects priority functions that excel on the training input data. The alternate input data likely exercises different paths of control flow—paths which may have been unused during training. Nonetheless, in every case, the application-specific priority function outperforms the baseline.

Figure 5 shows fitness improvements over generations. In many cases, Meta Optimization finds a superior priority function quickly, and finds only marginal improvements as the evolution continues. In fact, the baseline priority function is quickly obscured by GP-generated expressions. Often, the *initial* population contains at least one expression that outperforms the baseline. This means that by simply creating and testing 399 random expressions, we were able to find a priority function that outperformed Trimaran's for the given benchmark.

Once GP has discovered a decent solution, the search space and operator dynamics are such that most offspring will be worse, some will be equal and very few turn out to be better. This seems indicative of a steep hill in the solution space. In addition, multiple reruns using different initialization seeds reveal minuscule differences in performance. It might be a space in which there are many possible solutions associated with a given fitness.

5.4.2 General-Purpose Priority Functions

We divided the benchmarks in Table 5 into two sets[6]: a training set, and a test set. Instead of creating a priority

[6] We chose to train mostly on Mediabench applications because they compile and run faster than the Spec benchmarks.

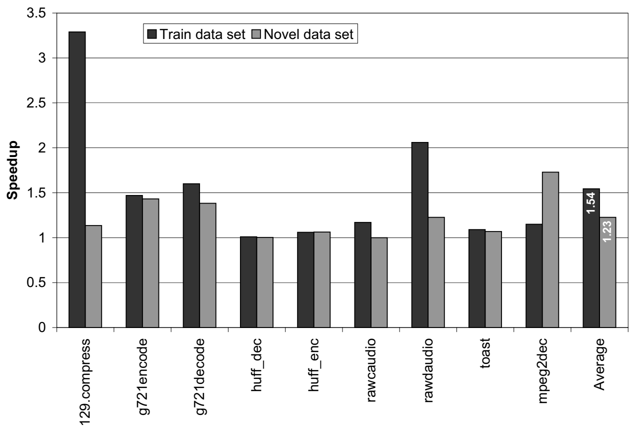

Figure 4: Hyperblock specialization. This graph shows speedups obtained by training on a per-benchmarks basis. The dark colored bars are executions using the same data set on which the specialized priority function was trained. The light colored bars are executions that use an alternate, or novel data set.

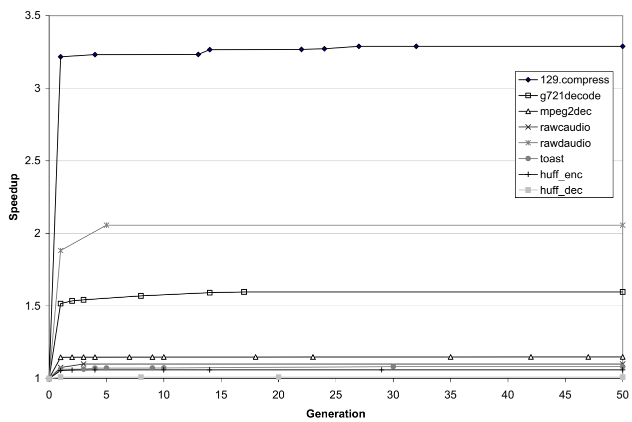

Figure 5: Hyperblock formation evolution. This figure graphs the best fitness over generations. For this problem, Meta Optimization quickly finds a priority function that outperforms Trimaran's baseline heuristic.

function for each benchmark, in this section we aim to find one priority function that works well for all the benchmarks in the training set. To this end, we evolve over the training set using dynamic subset selection [9].

Figure 6 shows the results of applying the single best priority function to the benchmarks in the training set. The dark bar associated with each benchmark is the speedup over Trimaran's base heuristic when the training input data is used. This data set yields a 44% improvement. The light bar shows results when novel input data is used. The overall improvement for this set is 25%.

It is interesting that, on average, the general-purpose priority function outperforms the application-specific priority function on the novel data set. The general-purpose solution is less susceptible to variations in input data because it was trained to be more general.

We then apply the resulting priority function to the benchmarks in the test set. The machine-learning community

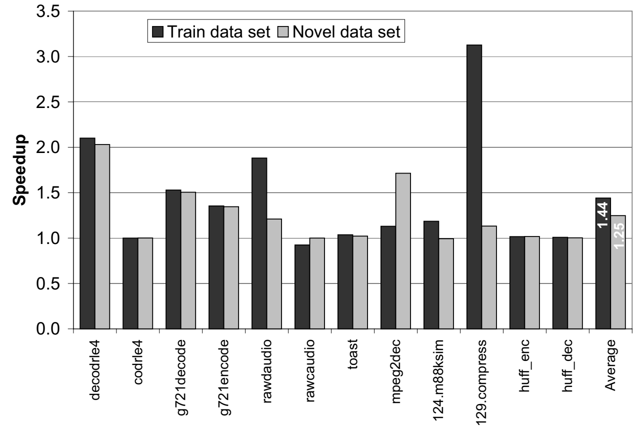

Figure 6: Training on multiple benchmarks. A *single* priority function was obtained by training over all the benchmarks in this graph. The dark bars represent speedups obtained by running the given benchmark on the same data that was used to train the priority function. The light bars correspond to a novel data set.

refers to this as cross validation. Since the benchmarks in the test set are not related to the benchmarks in the training set, this is a measure of the priority function's generality.

The results of the cross validation are shown in Figure 7. This experiment applies the best priority function on the training set to the benchmarks in the test set. The average speedup on the test set is 9%. In three cases (unepic, 023.eqntott, and 085.cc1) Trimaran's baseline heuristic marginally outperforms the GP-generated priority function. For the remaining benchmarks, the heuristic our system found is better.

5.4.3 The Best Priority Function

Figure 8 shows the best general-purpose priority function our system found for hyperblock selection. Because parsimony pressure favors small expressions, most of our system's solutions are readable. Nevertheless, the expressions presented in this paper have been hand simplified for ease of discussion.

Notice that some parts of the expression have no impact on the overall result. For instance, removing the sub-expression on line 2 will not affect the heuristic; the value is invariant to a scheduling region since the mean execution ratio is the same for all paths in the region. Such 'useless' expressions are called introns. It turns out that introns are actually quite useful for preserving good building blocks during crossover and mutation [13].

The conditional multiply statement on line 4 does have a direct effect on the priority function: it favors paths that do not have pointer dereferences (because the sub-expression in line 5 will always be greater than one). Pointers inhibit the effectiveness of the scheduler and other compiler optimizations, and thus dereferences should be penalized. The IMPACT group came to the exact same conclusion, though the extent to which they penalize dereferences differs [15].

The sub-expression on line 8 favors 'bushy' parallel paths, where there are numerous independent operations. This result is somewhat counterintuitive since highly parallel paths will quickly saturate machine resources. In addition, paths

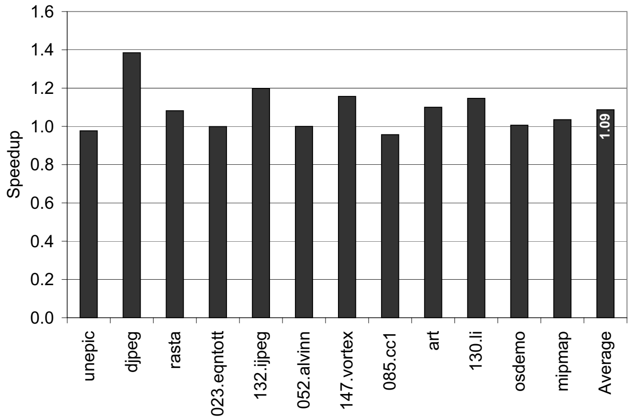

Figure 7: Cross validation of the general-purpose priority function. The best priority function found by training on the benchmarks in Figure 6 is applied to the benchmarks in this graph.

```
(1)  (add
(2)   (sub (mul exec_ratio_mean 0.8720) 0.9400)
(3)   (mul 0.4762
(4)        (cmul (not mem_hazard)
(5)         (mul 0.6727 num_paths)
(6)         (mul 1.1609
(7)          (add
(8)           (sub
(9)            (mul
(10)            (div num_ops dep_height) 10.8240)
(11)          exec_ratio)
(12)          (sub (mul (cmul has_unsafe_jsr
(13)                          predict_product_mean
(14)                          0.9838)
(15)                     (sub 1.1039 num_ops_max))
(16)               (sub (mul dep_height_mean
(17)                         num_branches_max)
(18)                    num_paths)))))))
```

Figure 8: The best priority function our system found for hyperblock scheduling.

with higher *exec_ratio*'s are slightly penalized, which also defies intuition.

The conditional multiply expression on line 12 penalizes paths with unsafe calls (*i.e.*, calls to subroutines that may have side effects). Once again this agrees with the IMPACT group's reasoning [15].

Because Trimaran is such a large and complicated system, it is difficult to know exactly why the priority function in Figure 8 works well. This is exactly the point of using a methodology like Meta Optimization. The bountiful complexities of compilers and systems are difficult to understand. Also worthy of notice is the fact that we get such good speedups, particularly on the training set, by changing such a small portion of the compiler.

The next section presents another case study, which we also test on Trimaran.

6. CASE STUDY II: REGISTER ALLOCATION

The importance of register allocation is well-known, so we will not motivate the optimization here. Many register allocation algorithms use cost functions to determine which variables to spill when spilling is required. For instance in priority-based coloring register allocation, the priority function is an estimate of the relative benefits of storing a given variable in a register [7].

Priority-based coloring first associates a *live range* with every variable. A live range is the composition of code segments (basic blocks), through which the associated variable's value must be preserved. The algorithm then prioritizes each live range based on the estimated execution savings of register allocating the associated variable:

$$savings_i = w_i \cdot (LDsave \cdot uses_i + STsave \cdot defs_i) \quad (2)$$

$$priority(lr) = \frac{\sum_{i \in lr} savings_i}{N} \quad (3)$$

Equation 2 is used to compute the savings of each code segment. $LDsave$ and $STsave$ are estimates of the execution time saved by keeping the associated variable in a register for references and definitions respectively. $uses_i$ and $defs_i$ represent the number of uses and definitions of a variable in block i. w_i is the estimated execution frequency for the block.

Equation 3 sums the savings over the N blocks that compose the live range. Thus, this priority function represents the savings incurred by accessing a register instead of resorting to main memory.

The algorithm then tries to assign registers to live ranges in priority order. Please see [7] for a complete description of the algorithm. For our purposes, the important thing to note is that the success of the algorithm depends on the priority function.

The priority function described above is intuitive— it assigns weights to live ranges based on the estimated execution savings of register allocating them. Nevertheless, our system finds functions that improve the heuristic by up to 11%.

6.1 Experimental Results

We collected these results using the same experimental setup that we used for hyperblock selection. We use Trimaran and we target the architecture described in Table 3. However, to more effectively stress the register allocator, we only use 32 general-purpose registers and 32 floating-point registers.

We modified Trimaran's Elcor register allocator by replacing its priority function (Equation 2) with an expression parser and evaluator. The register allocation heuristic described above essentially works at the basic block level. Equation 3 simply sums and normalizes the priorities of the individual basic blocks. For this reason, we stay within the algorithm's framework and leave Equation 3 intact.

For a more detailed description of our experiments with register allocation, including the features we extracted to perform them, please see [23].

6.1.1 Specialized Priority Functions

These results indicate that Meta Optimization works well, even for well-studied heuristics. Figure 9 shows speedups obtained by specializing Trimaran's register allocator for a given application. The dark bar associated with each application represents the speedup obtained by using the same input data that was used to specialize the heuristic. The light bar shows the speedup when the benchmark processes a novel data set.

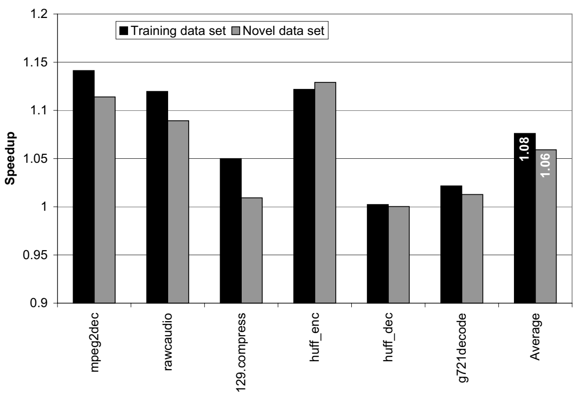

Figure 9: Register allocation specialization. This graph shows speedups obtained by training on a per-benchmarks basis. The dark colored bars are executions using the same data set on which the specialized priority function was trained. The light colored bars are executions that use a novel data set.

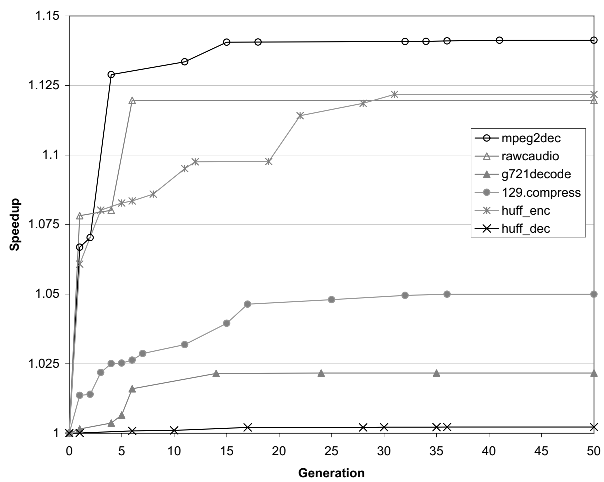

Figure 10: Register allocation evolution. This figure graphs fitness over generations. Unlike the hyperblock selection evolution, these fitnesses improve gradually.

Once again, it makes sense that the training input data outperforms the alternate input data. In the case of register allocation however, we see that the disparity between speedups on training and novel data is less pronounced than it is with hyperblock selection. This is likely because hyperblock selection is extremely data-driven. An examination of the general-purpose hyperblock formation heuristic reveals two dynamic factors (*exec_ratio* and *predict_product_mean*) that are critical components in the hyperblock decision process.

Figure 10 graphs fitness improvements over generations. It is interesting to contrast this graph with Figure 5. The fairly constant improvement in fitness over several generations seems to suggest that this problem is harder to optimize than hyperblock selection. Additionally, unlike the hyperblock selection algorithm, the baseline heuristic typically remained in the population for several generations.

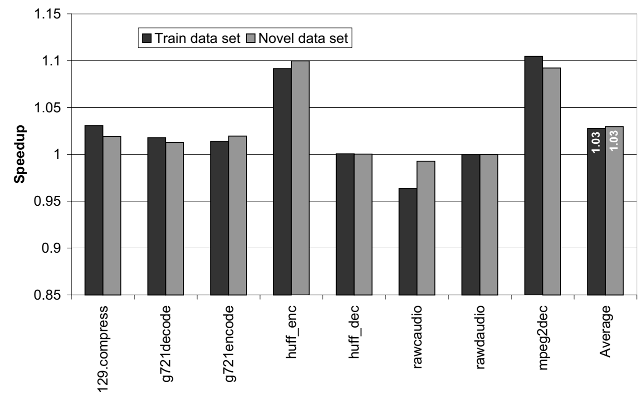

Figure 11: Training a register allocation priority function on multiple benchmarks. Our DSS evolution trained on all the benchmarks in this figure. The single best priority function was applied to all the benchmarks. The dark bars represent speedups obtained by running the given benchmark on the same data that was used to train the priority function. The light bars correspond to an alternate data set.

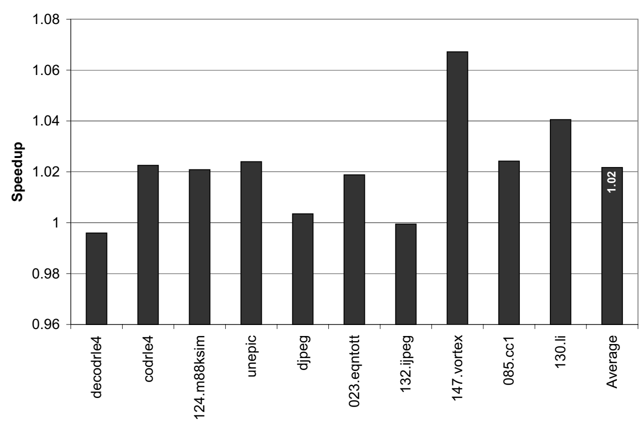

Figure 12: Cross validation of the general-purpose register allocation priority function. The best priority function found by the DSS run is applied to the benchmarks in this graph. Results from two target architectures are shown.

6.1.2 General-Purpose Priority Functions

Just as we did in Section 5.4.2, we divide our benchmarks into a training set and a test set[7]. The benchmarks in Figure 11 show the training set for this experiment. The figure also shows the results of applying the best priority function (from our DSS run) to all the benchmarks in the set. The dark bar associated with each benchmark is the speedup over Trimaran's baseline heuristic when using the training input data. The average for this data set is 3%. On a novel data set we attain an average speedup of 3%, which indicates that register allocation is not as susceptible to variations in input data.

Figure 12 shows the cross validation results for this ex-

[7] This experiment uses smaller test and training sets due to preexisting bugs in Trimaran. It does not correctly compile several of our benchmarks when targeting a machine with 32 registers.

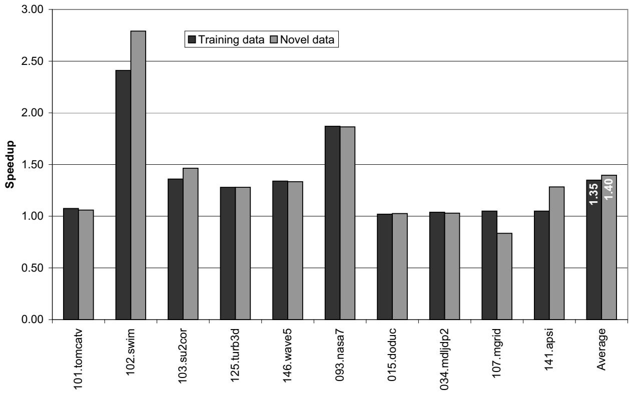

Figure 13: Prefetching specialization. This graph shows speedups obtained by training on a per-benchmarks basis. The dark colored bars are executions using the same data set on which the specialized priority function was trained. The light colored bars are executions that use a novel data set.

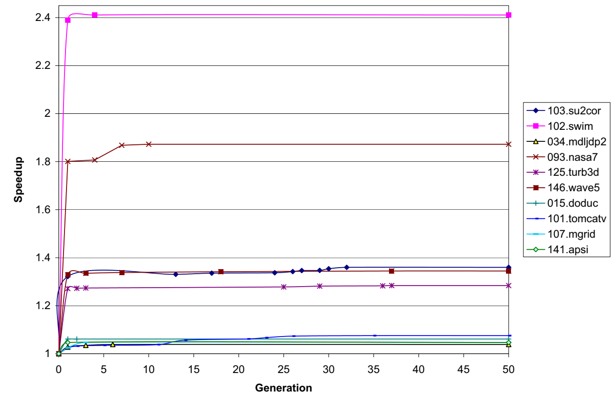

Figure 14: Prefetching evolution. This figure graphs fitness over generations. The baseline expression is quickly weeded out of the population.

periment. The figure shows the speedups (over Trimaran's baseline) achieved by applying the single best priority function to a set of benchmarks that were not in the training set. The learned priority function outperforms the baseline for all benchmarks except decodrle4 and 132.ijpeg. Although the overall speedup on the cross validation set is only 3%, this is an exciting result. Register allocation is well-studied optimization which our technique is able to improve.

7. CASE STUDY III: DATA PREFETCHING

This section describes another memory hierarchy optimization. Data prefetching is an optimization aimed at reducing the costs of long-latency memory accesses. By moving data from main memory into cache *before* it is accessed, prefetching can effectively reduce memory latencies.

However, prefetching can degrade performance in many cases. For instance, aggressive prefetching may evict useful data from the cache before it is needed. In addition, adding unnecessary prefetch instructions may hinder instruction cache performance and saturate memory queues.

The Open Research Compiler (ORC) [19] uses an extension of Mowry's algorithm [18] to insert prefetch instructions. ORC uses a priority function that assigns a Boolean confidence to prefetching a given address. Subsequent passes use this value to determine whether or not to prefetch the address. Currently, the priority function is simply based upon how well the compiler can estimate loop trip counts.

7.1 Experimental Setup

This case study is different from those already presented in two important ways. First, we collected the results of this section in the context of a real machine: we use the Open Research Compiler, and we target an Itanium I architecture. Just as with the previous two case studies, the fitness of an expression is the speedup over the baseline priority function. However, unlike simulated execution which is perfectly reproducible, real environments are inherently noisy. Even on an unloaded system, back-to-back runs of a program may vary.

Fortunately, GP can handle noisy environments, as long as the level of noise is smaller than attainable speedups using our technique. For the Itanium processor, this is indeed the case. Since it is a single threaded, statically scheduled processor, our measurements are fairly consistent; variations due to noise are well within the range of our attained speedups.

Another major divergence from the methodology employed in the last two case studies is the format of the priority function that we aim to optimize. Whereas the priority functions for register allocation and hyperblock formation are real-valued, the prefetching priority function is Boolean-valued. This case study emphasizes GP's flexibility.

As the ORC website recommends, we compile all benchmarks with `-O3` optimizations enabled, and we use profile-driven feedback. For additional details such as the features we extracted for this optimization, please see [23].

7.2 Experimental Results

Prefetching is known to be an effective technique for floating point benchmarks, and for this reason we train on various SPECFP benchmarks in this case study.

7.2.1 Specialized Priority Functions

Figure 13 shows the results of the ten different application-specialized priority functions. Closer examination of the GP solutions reveals that ORC overzealously prefetches and that by simply taming the compiler's aggressiveness, one can substantially improve performance (on this set of benchmarks). The GP solutions rarely prefetched. In fact, shutting off prefetching altogether achieves gains within 7% of the specialized priority functions.

Figure 14 graphs fitness over generation for the application-specific experiments. Just as with hyperblock selection, the baseline priority function has no impact on the final solutions— it is quickly obscured by superior expressions. As is the case with hyperblock selection, it appears that in many cases, GP solutions get 'stuck' in a local minimum in the solution space; the fitnesses stop improving early in the evolution. One plausible explanation for this is our use of parsimony pressure in the selection process. For application-specific evolutions, it is often the case that very small expressions work well. While these small expressions are effective,

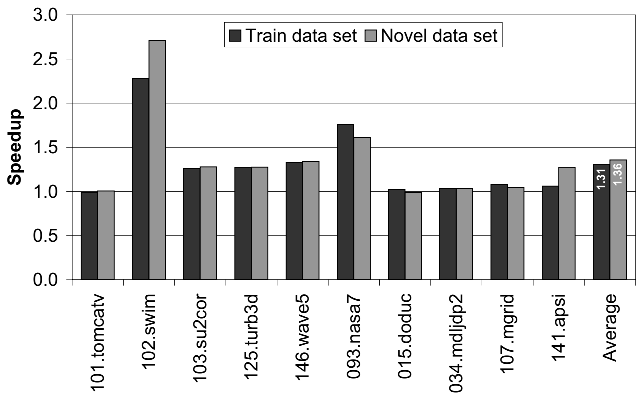

Figure 15: Training a prefetching priority function on multiple benchmarks. Our DSS evolution trained on all the benchmarks in this figure. The single best priority function was applied to all the benchmarks. The dark bars represent speedups obtained by running the given benchmark on the same data that was used to train the priority function. The light bars correspond to an alternate data set.

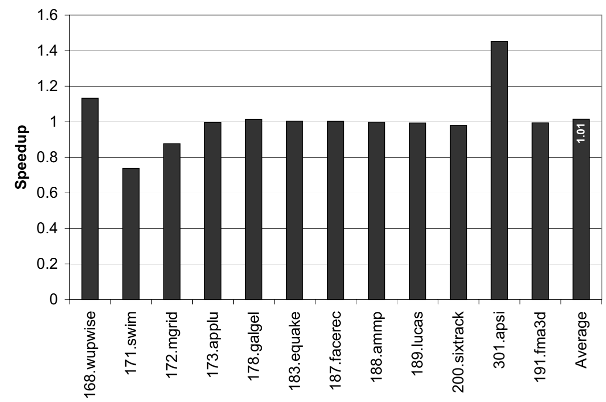

Figure 16: Cross validation of the general-purpose prefetching priority function on SPEC2000. The best priority function found by the DSS run is applied to the benchmarks in this graph. Results from two target architectures are shown.

they limit the genetic material available to the crossover operator. Furthermore, since we always keep the best expression, the population soon becomes inbred with copies of the top expression. Future work will explore the impact of parsimony pressure and elitism.

7.2.2 General-Purpose Priority Functions

The performance of the best DSS-generated prefetching priority function is shown in Figure 15. The priority function was trained on the same benchmarks in the figure, which are a combination of SPEC92 and SPEC95 floating point benchmarks. Data prefetching, like hyperblock selection, is extremely data-driven. By applying the same input data that we used to train the priority function we achieve a 31% speedup. Somewhat surprisingly, the novel input data set achieves a better speedup of 36%. Because the priority function learned to prefetch infrequently, it is simply the case that the novel data set is more sensitive to prefetching than the training data set is.

Figure 16 shows the cross validation results for this optimization, and prompts us to mention a caveat of our technique. GP's ability to identify good general-purpose solutions is based on the benchmarks over which they are evolved. For the SPEC92 and SPEC95 benchmarks that were used to train our general-purpose heuristic, aggressive prefetching was debilitating. However, for a couple of benchmarks in the SPEC2000 floating point set, we see that aggressive prefetching is desirable. Thus, unless designers can assert that the training set provides adequate problem coverage, they cannot completely trust GP-generated solutions.

8. RELATED WORK

Many researchers have applied machine-learning methods to compilation, and therefore, only the most relevant works are cited here.

Calder et al. used supervised learning techniques to fine-tune static branch prediction heuristics [5]. They employ neural networks and decision trees to search for effective static branch prediction heuristics. While our methodology is similar, our work differs in several important ways. Most importantly, we use unsupervised learning, while they use supervised learning.

Unsupervised learning is used to capture inherent organization in data, and thus, only input data is required for training. Supervised learning attempts to match training inputs with known outcomes, called *labels*. This means that their learning techniques rely on knowing the optimal outcome, while ours does not[8]. In their case determining the optimal outcome is trivial— they simply run the benchmarks in their training set and note the direction that each branch favors. In this sense, their method is simply a classifier: classify the data into two groups, either taken or not-taken. Priority functions cannot be classified in this way, and thus they demand an unsupervised method such as ours.

We also differ in the end goal of our learning techniques. They use misprediction rates to guide the learning process. While this is a perfectly valid choice, it does not necessarily reflect the bottom line: execution time.

Monsifrot et al. use a classifier based on decision tree learning to determine which loops to unroll [17]. Like [5], this supervised methodology relies on extracting labels, which is not only difficult, in many cases it is simply not feasible.

Cooper et al. use genetic algorithms to solve compilation phase ordering problems [8]. Their technique is quite effective. However, like other related work, they evolve the application, not the compiler[9]. Thus, their compiler iteratively evolves *every* program it compiles. By evolving compiler heuristics, and not the applications themselves, we need only apply our process once as shown in Section 5.4.2.

The COGEN(t) compiler creatively uses genetic algorithms to map code to irregular DSPs [11]. This compiler, though interesting, also evolves on a per-application basis. Nonetheless, the compile-once nature of DSP applications may warrant the long, iterative compilation process.

[8] This is a strict requirement both for decision trees and the gradient descent method they use to train their neural network.

[9] However, they were able to manually construct a general-purpose sequence using information gleaned from their application-specific evolutions.

9. CONCLUSION

Compiler developers have always had to contend with complex phenomenon that are not easy modeled. For example, it has never been possible to create a useful model for all the input programs the compiler has to optimize. However until recently, most architectures— the target of compiler optimizations— were simple and analyzable. This is no longer the case. A complex compiler with multiple interdependent optimization passes exacerbates the problem. In many instances, end-to-end performance can only be evaluated empirically.

Optimally solving NP-hard problems is not practical even when simple analytical models exist. Thus, heuristics play a major role in modern compilers. Borrowing techniques from the machine-learning community, we created a general framework for developing compiler heuristics. We advocate a machine-learning based methodology for automatically learning effective compiler heuristics.

The techniques presented in this paper show promise, but they are still in their infancy. For many applications our techniques found excellent application-specific priority functions. However, the disparity in some cases between the application-specific performance and the general-purpose performance tells us that our techniques can be improved.

We also note disparities between the performance of training set applications and the cross validation performance. In some cases our solutions *overfit* the training set. If compiler developers use our technique but only train using benchmarks on which their compiler will be judged, the generality of their compiler may actually be reduced.

Our fledgling research has a few shortcomings that future work will address. For instance, the success of any learning algorithm hinges on selecting the right features. We will explore techniques that aid in extracting features that best reflect program variability. While genetic programming is well-suited to our application, it too has shortcomings. The overriding goal of our research is to free humans from tedious parameter tweaking. Unfortunately, GP's success is dependent on parameters such as population size and mutation rate, and finding an adequate solution relies on some experimentation (which fortunately can be performed with a minimal amount of user interaction). Future work will experiment with different learning techniques.

We believe the benefits of using a system like ours far outweighs the drawbacks. While our techniques do not always achieve large speedups, they do reduce design complexity considerably. Compiler writers are forced to spend a large portion of their time designing heuristics. The results presented in this paper lead us to believe that machine-learning techniques can optimize heuristics at least as well human designers. We believe that automatic heuristic tuning based on empirical evaluation will become prevalent, and that designers will intentionally expose algorithm policies to facilitate machine-learning optimization.

A toolset that can be used to evolve compiler heuristics will be available at:

`http://www.cag.lcs.mit.edu/metaopt`

10. ACKNOWLEDGMENTS

We would like to thank the PLDI reviewers for their insightful comments. In general we thank everyone who helped us with this paper, especially Kristen Grauman, Michael Gordon, Sam Larsen, William Thies, Derek Bruening, and Vladimir Kiriansky. Finally, thanks to Michael Smith and Glenn Holloway at Harvard University for lending us their Itanium machines in our hour of need. This work is funded by DARPA, NSF, and the Oxygen Alliance.

11. REFERENCES

[1] S. G. Abraham, V. Kathail, and B. L. Deitrich. Meld Scheduling: Relaxing Scheduling Constaints Across Region Boundaries. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, pages 308–321, 1996.

[2] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann, 1998.

[3] D. Bernstein, D. Goldin, and M. G. et. al. Spill Code Minimization Techniques for Optimizing Compilers. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258–263, 1989.

[4] D. Bourgin. *Losslessy compression schemes*. http://hpux.u-aizu.ac.jp/hppd/hpux-/Languages/codecs-1.0/.

[5] B. Calder, D. G. ad Michael Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-Based Static Branch Prediction Using Machine Learning. In *ACM Transactions on Programming Languages and Systems (ToPLaS-19)*, volume 19, 1997.

[6] P. Chang, D. Lavery, S. Mahlke, W. Chen, and W. Hwu. The Importance of Prepass Code Scheduling for Superscalar and Superpipelined processors. In *IEEE Transactions on Computers*, volume 44, pages 353–370, March 1995.

[7] F. C. Chow and J. L. Hennessey. The Priority-Based Coloring Approch to Register Allocation. In *ACM Transactions on Programming Languages and Systems (ToPLaS-12)*, pages 501–536, 1990.

[8] K. Cooper, P. Scheilke, and D. Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. In *Languages, Compilers, Tools for Embedded Systems*, pages 1–9, 1999.

[9] C. Gathercole. *An Investigation of Supervised Learning in Genetic Programming*. PhD thesis, University of Edinburgh, 1998.

[10] P. B. Gibbons and S. S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. In *Proceedings of the ACM Symposium on Compiler Construction*, volume 21, pages 11–16, 1986.

[11] G. W. Grewal and C. T. Wilson. Mappping Reference Code to Irregular DSPs with the Retargetable, Optimizing Compiler COGEN(T). In *International Symposium on Microarchitecture*, volume 34, pages 192–202, 2001

.

[12] M. Kessler and T. Haynes. Depth-Fair Crossover in Genetic Programming. In *Proceedings of the ACM Symposium on Applied Computing*, pages 319–323, February 1999.

[13] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. In *International Symposium on Microarchitecture*, volume 30, pages 330–335, 1997.

[15] S. A. Mahlke. *Exploiting instruction level parallelism in the presence of branches*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 1996.

[16] S. A. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *International Symposium on Microarchitecture*, volume 25, pages 45–54, 1992.

[17] A. Monsifrot, F. Bodin, and R. Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Artificial Intelligence: Methodology, Systems, Applications*, pages 41–50, 2002.

[18] T. C. Mowry. *Tolerating Latency through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Department of Electrical Engineering, 1994

.

[19] Open Research Compiler. *http://ipf-orc.sourceforge.net*.

[20] E. Ozer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register Filee Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-24)*, pages 308–315, 1998.

[21] J. C. H. Park and M. S. Schlansker. On Predicated Execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, 1991.

[22] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-24)*, November 1994.

[23] M. Stephenson, S. Amarasinghe, U.-M. O'Reilly, and M. Martin. Compiler Heuristic Design with Machine Learning. Technical Report TR-893, Massachusetts Institute of Technology, 2003.

[24] Trimaran. *http://www.trimaran.org*.

[25] N. Warter. *Modulo Scheduling with Isomorphic Control Transformations*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 1993.