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 an optimization's solution space. We implemented Meta Optimization on top of Trimaran [20] to test its efficacy. By 'evolving' Trimaran's hyperblock selection optimization for a particular benchmark, our system achieves impressive speedups. Application-specific heuristics obtain an average speedup of 23% (up to 43%) for the applications in our suite. Furthermore, by evolving a compiler's heuristic over several benchmarks, we can create effective, general-purpose compilers. The best general-purpose heuristic our system found improved Trimaran's hyperblock selection algorithm by an average of 25% on our training set, and 9% on a completely unrelated test set. We further test the applicability of our system on Trimaran's priority-based coloring register allocator. For this well-studied optimization we were able to specialize the compiler for individual applications, achieving an average speedup of 6%.

Keywords

machine learning, priority functions, genetic programming, compiler heuristics

1. INTRODUCTION

Compiler writers have a difficult task. They are expected to create effective and inexpensive solutions to NP-hard problems such as instruction scheduling and register allocation. This problem is complicated by the advent of intractably complex computer architectures.

Since it is impossible to create a simple model that captures the intricacies of modern architectures, compiler writers rely on inaccurate abstractions. Such models are based upon many assumptions, and thus may not even properly simulate first-order effects. In addition, changing an architecture likely requires substantial changes to the compiler.

Faced with these challenges, developing a compiler is more of a black art than a science. Most of the optimizations that a compiler performs are NP-hard. Compilers often perform several optimizations with competing and conflicting goals. Getting everything to mesh nicely is a daunting task.

Compilers cannot afford to optimally solve NP-hard problems. Therefore compiler writers devise clever heuristics that 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).

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 choices along which a compiler algorithm can proceed.

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 [4]. 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.

This key insight into compiler heuristics 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

MIT/LCS Technical Memo,
LCS-TM-XXX,
December 3, 2002.

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. Alternatively, it can be used to find a general-purpose function that works well for a broad range of applications. Experimental results show an average of 23% improvement (over well-known baseline heuristics) when specializing for individual applications. By 'evolving' over a set of benchmarks, our system found a general-purpose priority function that achieves an average speedup of 25% when applied to the benchmarks on which it was trained. Demonstrating its generality, the priority function achieves an average speedup of 9% on a completely unrelated set of benchmarks.

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 a compiler, 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 presents two case studies that we use to determine the efficacy of Meta Optimization. Section 5 discusses our experimental framework. We present encouraging results in Section 6. Section 7 discusses related work, and finally Section 8 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 [11]. 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 blindly uses 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 two 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 [17]. They note that the choice of priority function has a "strong effect on the schedule." They also investigate five different priority functions [17].

- **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 [19], Rau states, "As is the case for acyclic list scheduling, there is a limitless number of priority functions that can be devised for modulo scheduling." Rau describes the tradeoffs involved when considering scheduling priorities.

- **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 [14]. 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 are not as comprehensible.

Like other evolutionary algorithms, GP is loosely patterned on Darwinian evolution. GP maintains a population of parse trees [14]. 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

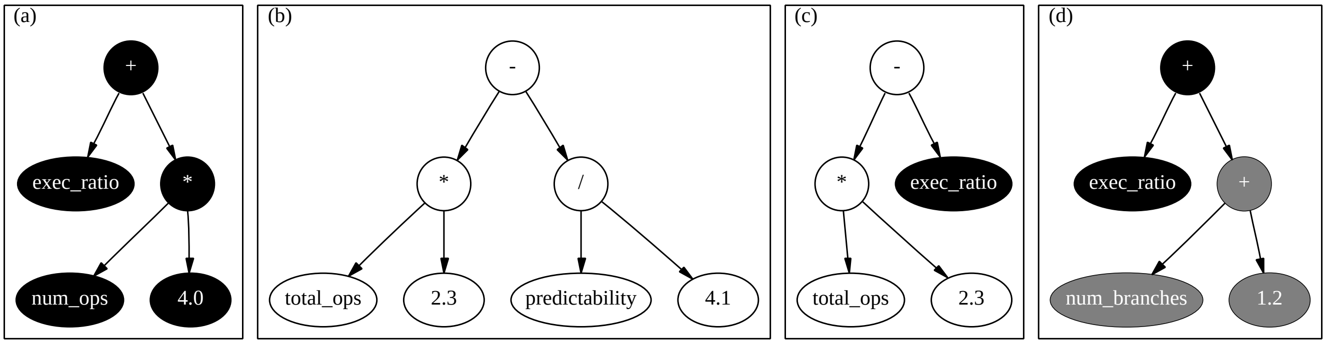

Figure 1: GP Genomes. Part (a) and (b) show examples of GP genomes. Part (c) provides an example of a random crossover of the genomes in (a) and (b). Part (d) shows a mutation of the genome in part (a).

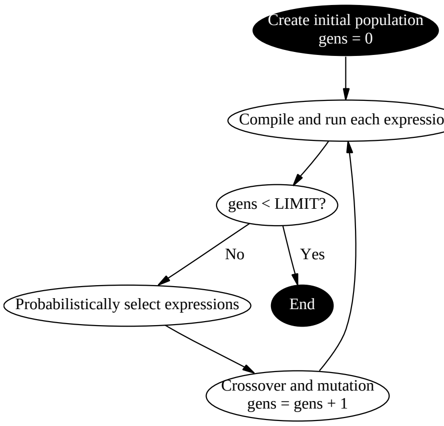

Figure 2: Flow of genetic programming. Genetic programming (GP) initially creates a population of expressions. Each expression is then assigned a fitness, which is a measure of how well it satisfies the end goal. In our case, fitness is proportional to the execution time of the compiled application(s). 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.

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. The fitness is a measure of the expression's effectiveness. 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 [14, p. 109]. Parsimonious expressions are aligned with our philosophy of using GP as a tool for compiler writers and architects to identify causal variables 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 [14]. 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, p. 242].

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 [10]. The technique—called dynamic subset selection (DSS)— essentially trains on subsets of the training programs, concentrating more effort on programs that perform poorly compared to the baseline heuristics.

The next section describes two heuristics that we will use as case studies.

4. TWO CASE STUDIES

This section motivates and discusses two popular compiler heuristics that we use as case studies throughout the remainder of the paper: IMPACT's hyperblock formation algorithm [15] and priority-based register allocation [8]. Section 5 quantitatively shows that small changes to the priority

[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 root nodes over half of the time. We employ depth-fair crossover, which equally weighs each level of the tree [13].

functions associated with these heuristics have a tremendous impact on the performance of compiled code.

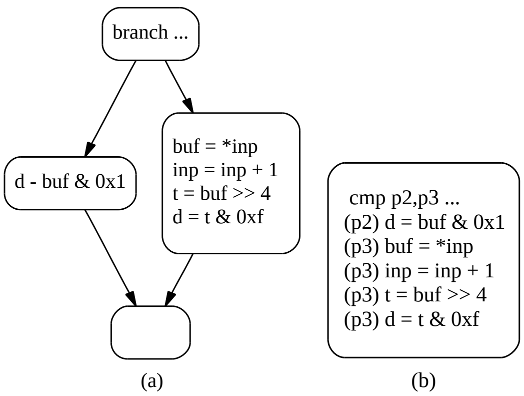

Figure 3: Control flow *v.* predicated execution. Part (a) shows a segment of control-flow that demonstrates a simple if-then-else statement. As is typical with multimedia and integer applications, there are few instructions per basic block in the example. Part (b) is the corresponding predicated hyperblock. 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.

4.1 Case Study I: Hyperblock Formation

Architects have proposed two noteworthy methods for decreasing the costs[2] associated with control transfers: 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.

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.

In the following list we give a brief overview of several criteria that are useful to consider when forming hyperblocks. 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, pp. 146,148]. 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.

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

- **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. We now discuss the heuristic employed by the IMPACT compiler [15, 16]. This heuristic was later adopted by the SGI Pro64 compiler.

Because loop-backedges are not predicatable, the IMPACT compiler attempts to coalesce them into a single back-edge [15]. This transformation creates more predicatable paths. The algorithm then identifies acyclic paths of control that are suitable for hyperblock inclusion. Park and Schlansker detail this portion of the algorithm in [18]. 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. The variable *exec_ratio*, is a measure of how frequently the path in question executes. In other words, 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*).

In Section 6 we show that this heuristic can be drastically improved simply by replacing its priority function with one determined automatically using machine-learning techniques.

4.2 Case Study II: Priority-Based Coloring 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 live range in a register [8]. A live range is defined as a contiguous group of basic blocks in which a variable is live. [8]'s approach creates live ranges and prioritizes them according to the following equations:

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

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

Here, a collection of basic blocks compose a live range (lr). $LDsave$ and $STsave$ are estimates of the execution time saved by keeping a variable's live range in a register for references and definitions respectively. $uses_i$ and $defs_i$ represent the number of uses and definitions of a variable in basic block i. N is the number of basic blocks in the live range, and w_i is an execution frequency estimate for block i.

The algorithm then tries to assign registers to live ranges in priority order. Please see [8] 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, as we will show in Section 6, our system found functions that improve the heuristic by up to 11%.

The next section describes the experimental framework we use to 'evolve' priority functions for the two optimizations covered in this section.

5. EXPERIMENTAL FRAMEWORK

While the priority functions described in the last section are intuitive, they are based on a simple model that does not capture the full essence of a complex system. While this may seem counterintuitive, Section 6 will show that variations in the priority function can have a tremendous influence on performance.

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. This restriction is not constraining; even small changes to the priority function can drastically improve (or diminish) performance.

Our infrastructure is built upon Trimaran [20]. Trimaran is an integrated compiler and simulator for a parameterized EPIC architecture. Table 1 details the specific architecture over which we evolved. This model is similar to Intel's Itanium® architecture.

We tackled the two case studies separately, starting with hyperblock formation. 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 variables that might be important for creating effective priority functions. Table 2 describes these variables.

The hyperblock selection algorithm passes the variables in the table as parameters to the expression evaluator. For instance, if an expression contains a reference to `dep_height`, the selection algorithm will evaluate the expression with the value of the path's dependence height. Most of the characteristics in Table 2 were already available in IMPACT. The priority function in Equation 1 has a local scope. We

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 1: Architectural characteristics. This table describes the EPIC architecture over which we evolved. This approximates the Intel Itanium architecture. For the register allocation problem, we used 32 general-purpose registers and 32 floating-point registers.

added the minimum, maximum, mean, and standard deviation of all path-specific characteristics, which encapsulates some global knowledge.

We modified the compiler's profiler to extract branch predictability statistics. In addition, we added a 2-bit dynamic branch predictor to the simulator.

For the second case study, 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 in Section 4.2 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.

Table 3 shows characteristics relating to the priority-based coloring register allocator. To more effectively stress the register allocator, we only use 32 general-purpose registers and 32 floating-point registers.

We built the iterative framework of Figure 2 around Trimaran. The initial population consists of randomly initialized expressions, as well as Trimaran's original priority function (Equation 1 for hyperblock formation, and Equation 2 for register allocation).

The results presented in this paper use total execution time (reported by the Trimaran system) to assign fitness. This approach rewards 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. Future work will experiment with different fitness evaluations.

Table 4 shows the 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. Some of the primitives we included are not present in the final solutions, which suggests that they might be useless components of a priority function. In future work, we will experiment with different primitive sets.

Characteristic	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	Path predictability obtained by simulating a branch predictor (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 2: Characteristics that might affect hyperblock selection. 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.

6. RESULTS

This section discusses the results we obtained with our GP system. Our system evolved each of the compiler priority functions (genomes) for 50 generations. Our algorithm maintains a population of 400 expressions. It *randomly* replaces 22% of the population every generation. Only the *single* best expression is guaranteed survival. The system creates new expressions via the crossover operator discussed in Section 3. The mutation operator described in the same section mutates roughly 5% of the new expressions. We use tournament selection with a tournament size of 7 to choose expressions for crossover. This setting causes moderate selection pressure. Table 5 summarizes the GP parameters.

For both case studies, we enabled the following Trimaran compiler optimizations: function inlining, loop unrolling, backedge coalescing, acyclic global scheduling [7], modulo scheduling [21], hyperblock formation, register allocation, machine-specific peephole optimization, and several classic optimizations.

Table 6 shows the benchmarks this section surveys. All of the Trimaran certified benchmarks are included in the table[3] [20]. Our suite also includes most of the Mediabench benchmarks. The build process for ghostscript proved too

[3] We could not get 134.perl to execute correctly, though [20] certified it.

Characteristic	Description
spill_cost	The estimated cost of spilling this range to memory. See Equation 2.
region_weight	Number of times the basic block was executed (from profile).
live_ops	The number of live operations in the block.
num_calls	The number of procedure calls in a basic block.
callee_benefit	The callee's 'benefit' of allocating the range.
caller_benefit	The caller's 'benefit' of allocating the range.
def_num	The number of definitions in the block.
use_num	The number of uses in the block.
STsave	Estimate of the execution time saved by keeping a definition in a register.
LDsave	Estimate of the execution time saved by keeping a reference in a register.
has_single_ref	If the block has a single reference this returns *true*, otherwise it returns *false*.
is_pass_through	If the number of live references in the block is greater than 0, return *true*, otherwise return *false*.
ref_op_count	The number of references in the block.
reg_size	The number of registers available for the register class of the live range.
forbidden_regs	The number of registers that are not available to the live range (because it interferes with an allocated live range).
GPR, *FPR*, *PR*	Returns *true* if the live range belongs to the class GPR, FPR, or PR respectively; returns *false* otherwise.

Table 3: Characteristics that might affect register allocation.

difficult to compile. We exclude the remainder of the Mediabench applications because the Trimaran system does not compile them correctly[4].

6.1 Hyperblock Formation

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

6.1.1 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 when alternate input data is used.

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. However, the alternate input data likely exercises different paths of control flow—paths which may have been unused during training.

Figure 5 shows fitness improvements over generations. In many cases, Meta Optimization finds a superior priority function quickly, and finds only marginal improvements as

[4] We exclude cjpeg, the complement of djpeg, because it does not execute properly with some priority functions.

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 $\{true, false\}$	(bconst $\{true, false\}$)
Returns Boolean value of *arg* from environment	(barg *arg*)

Table 4: 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.

Parameter	Setting
Population size	400 expressions
Number of generations	50 generations
Generational replacement	22 expressions
Mutation rate	5%
Tournament size	7
Fitness	Total execution time for compiler specialization. Total execution time normalized on an input-program basis for DSS-style training.

Table 5: GP parameters. This table shows the GP parameters we used to collect the results in this section.

the evolution continues. In fact, the baseline priority function is often 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 homed in on 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.

Benchmark	Suite	Description
codrle4 decodrle4	See [5]	RLE type 4 encoder/decoder.
huff_enc huff_dec	See [5]	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 libraray 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 6: Benchmarks used. The set includes applications from the SpecInt, SpecFP, Mediabench benchmark suites, and a few miscellaneous programs.

6.1.2 *Finding General-Purpose Functions*

We divided the benchmarks in Table 6 into two sets[5]: a training set, and a test set. We evolve over the training set using dynamic subset selection [10]. We then apply the resulting priority function to the benchmarks in the test set. The machine-learning community 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.

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 alternate 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 for the reference data set. The general-purpose solution is less succeptible to variations in input data precisely because it is more generally applicable.

The results of the cross validation are shown in Figure 7. This experiment applies the best priority function on the test set to the benchmarks in the training set. The average speedup over the test set is 9%. In three cases (unepic, 023.eqntot, 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] We chose to train mostly on Mediabench applications because they compile and run faster than the Spec benchmarks. However, we randomly chose two Spec benchmarks for added coverage.

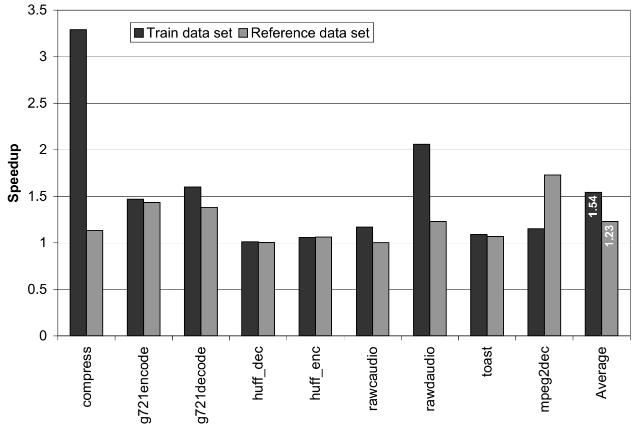

Figure 4: Hyperblock specialization. This graph shows speedups obtained by training on a per-benchmarks basis. The dark colored bars are executions using the same data set on which the specialized priority function was trained. The light colored bars are executions that use an alternate data set.

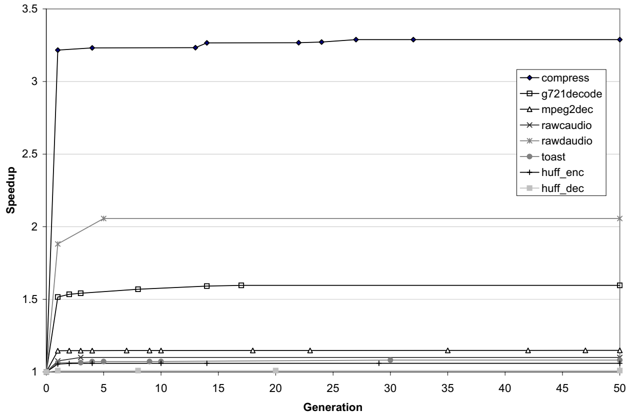

Figure 5: Hyperblock formation evolution. This figure graphs fitness over generations. For this problem, Meta Optimization often finds a priority function that outperforms Trimaran's baseline heuristic.

6.2 Register Allocation

Preliminary results indicate that Meta Optimization works well, even for well-studied heuristics. Figure 8 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 an alternate data is used.

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 difference between the two is less pronounced. 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 $predictability$) that are critical components in the hyperblock decision process.

Figure 9 graphs fitness improvements over generations. It is interesting to contrast this graph with Figure 5. The

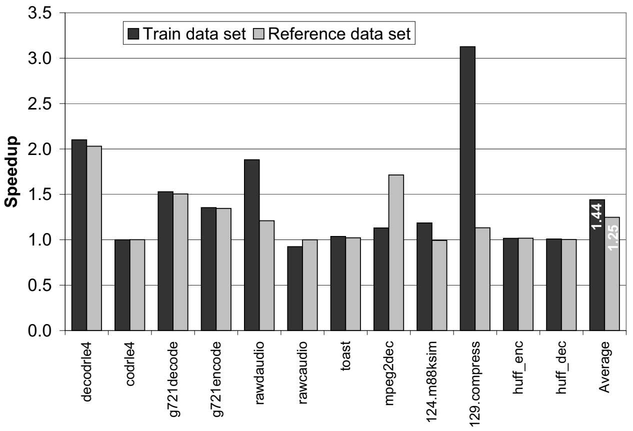

Figure 6: Training on multiple benchmarks. A *single* priority function was obtained by training over all the benchmarks in this graph. The dark bars represent speedups obtained by running the given benchmark on the same data that was used to train the priority function. The light bars correspond to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

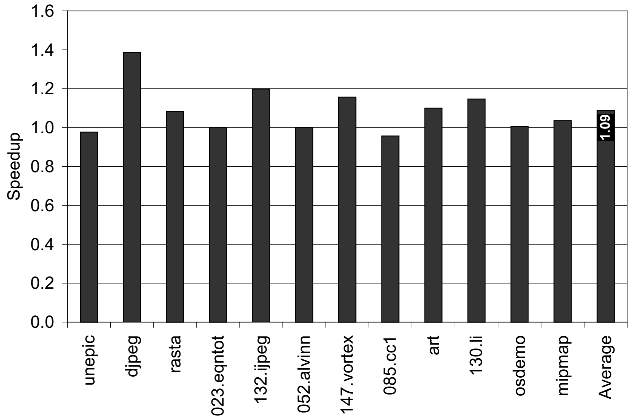

Figure 7: Cross validation of the general-purpose priority function. The best priority function found by training on the benchmarks in Figure 6 is applied to the benchmarks in this graph.

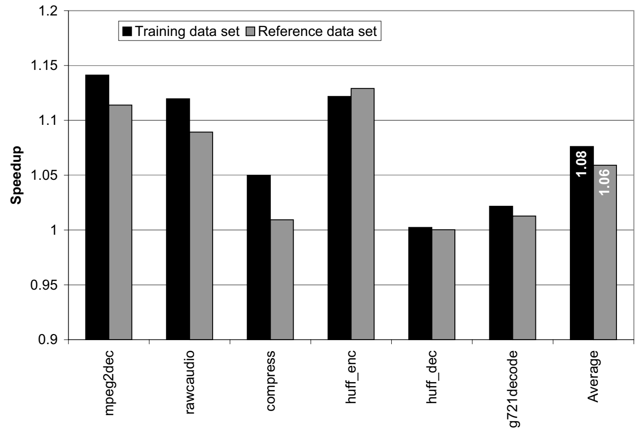

Figure 8: Register allocation specialization. This graph shows speedups obtained by training on a per-benchmarks basis. The dark colored bars are executions using the same data set on which the specialized priority function was trained. The light colored bars are executions that use an alternate data set.

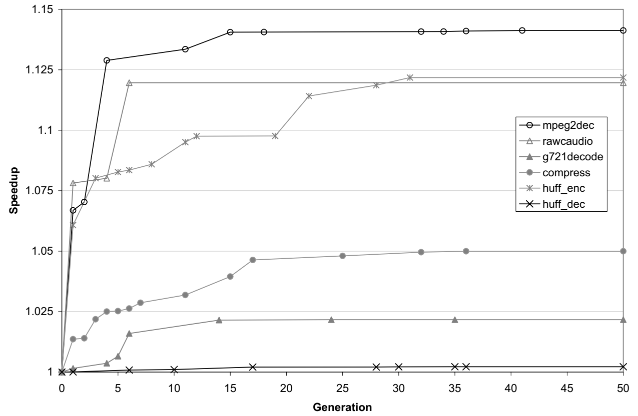

Figure 9: Register allocation evolution. This figure graphs fitness over generations. Unlike the hyperblock selection evolution, these fitnesses converge slowly.

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 was typically retained (*i.e.*, it remained in the population) for several generations.

We were unable to evolve a general solution for register allocation in time for this submission. With 20 processors, these results take about 6 days to collect.

7. 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 [6]. They employ two learning techniques — 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 learns to match training inputs with known outcomes. This means that their learning techniques rely on knowing the optimal outcome[6], while ours does not. 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

[6]In fact, this is a strict requirement both for decision trees and the gradient descent method they use to train their neural network.

reflect the bottom line: execution time.

Cooper et. al use genetic algorithms to solve compilation phase ordering problems [9]. Their technique is quite effective. However, like other related work, they evolve the application, not the compiler. 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.

The COGEN(t) compiler creatively uses genetic algorithms to map code to irregular DSPs [12]. 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.

Beaty's instruction scheduler based on genetic algorithms is not only application-specific, it is also data-specific [3]. By not enforcing correctness, Beaty's algorithm may work for the data set on which it evolves, but not for an alternate data set.

8. CONCLUSION

Compiler developers have always had to contend with complex phenomenon that are not easy modeled. For example, it is not 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 propose a genetic programming based methodology for automatically learning effective priority functions.

This paper investigated two such heuristics— hyperblock selection and register allocation. Our technique identified more effective priority functions than the baseline functions against which we compared. For the hyperblock formation optimization, we achieved an application-specific speedup of 23%. Furthermore, we found a general-purpose priority function that lead to a 25% improvement on the benchmarks on which it was trained, and a 9% improvement on completely unrelated applications. For the register allocation problem, our technique discovered application-specific priority functions that yield a 6% overall improvement.

Compiler writers are forced to spend a large portion of their time tweaking heuristics. We believe that automatic heuristic tuning based on empirical evaluation will become prevalent.

9. 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] S. J. Beaty. Genetic Algorithms and Instruction Scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, November 1991.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[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] E. Ozer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register Filee Microarchitectures.

[18] J. C. H. Park and M. S. Schlansker. On Predicated

Execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, 1991.

[19] 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.

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

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