

# Selector: A Language Construct for Developing Dynamic Applications

Pedro C. Diniz and Bing Liu

University of Southern California / Information Sciences Institute  
4676 Admiralty Way, Suite 1001  
Marina del Rey, California, 90292  
{pedro,bliu}@isi.edu

**Abstract.** Fitting algorithms to meet input data characteristics and/or a changing computing environment is a tedious and error prone task. Programmers need to deal with code instrumentation details and implement the selection of which algorithm best suits a given data set. In this paper we describe a set of simple programming constructs for C that allows programmers to specify and generate applications that can select at run-time the best of several possible implementations based on measured run-time performance and/or algorithmic input values. We describe the application of this approach to a realistic linear solver for an engineering crash analysis code. The preliminary experimental results reveal that this approach provides an effective mechanism for creating sophisticated dynamic application behavior with minimal effort.

## 1 Introduction

The best algorithm for a given computation differs widely depending on the characteristics of its input data and of features of the target architecture. For example there are several sorting algorithms that perform very well when the distribution of the keys is uniform (*e.g.*, *quicksort*) while other algorithms perform much better when the keys either have known boundary properties (*e.g.*, *bucket sort*) or are heavily modal (*e.g.*, *merge sort*). In other scenarios the environment characteristics, rather than the input characteristics impact the choice of algorithm more deeply. For instance, in the context of distributed applications it is possible to trade off computation with communication or simply to offload some of the computation to other nodes if the available bandwidth is adequate. Overall programmers would like to choose a particular implementation among a set of possible alternative implementations of the same functionality depending on input data characteristics, environment conditions or both. While in some cases it is possible to characterize the exact set of conditions for which each alternative implementation should be used (*e.g.*, sorting), in general, programmers must rely on observed behavior to decide which implementation performs best. For scenarios where the choice of which algorithm implementation depends on environment conditions programmers must manually instrument the code with

calls to a run-time-system and manually encode the strategies to dynamically select the appropriate algorithm implementation. Other than being cumbersome and error-prone, the resulting code is complex and hard to port or maintain. The approach proposed in this paper advocates extending an imperative programming language such as C/C++ with a modest set of programming constructs allowing programmers to specify the dynamic behavior of a set of alternative implementations for the same functionality. Programmers use a *selector* construct to associate several code variant implementations of the same function name and to define what the switching policy between the alternative code variants is. Programmers also specify which set of environment variables should be observed for which code variant and associate a *cost* and (optionally) a *probe* function with each code variant. The implementation uses the *probe* and *cost* functions to evaluate, rank and choose the best available variant. Because probing and selecting among a potentially large number of code variants can incur non-negligible overheads the *selector* construct provides a *trigger* function that can disable the probing of code variants for a specific number of invocations or until a relevant environment event occurs. *Trigger* functions provide a powerful mechanism to control the amount of probing overhead and encode the relevant environment conditions under which the alternative variant should be reevaluated. This paper makes the following specific contributions:

- Describes the *selector* construct - a modest set of language extensions for adaptive programming for imperative programming languages.
- Describes a particular implementation of the *selector* to C and outlines a source-to-source code generation scheme for C/C++.
- Presents results of the application of the *selector* construct to a sophisticated linear solver from a real engineering code.

While it is true that programmers can manually implement the functionality of the *selector*, there are several benefits to the approach outlined in this paper. First, it is automated. Programmers are not required to engage in low-level error-prone instrumentation of their codes. Second, the semantic gap between the *selector* semantics and the generated C code is not wide, thereby avoiding programmers second-guessing what the selector code will do.

We see the *selector* as a powerful tool for application and/or library developers whose needs are beyond what optimizing compilers can currently perform. The *selector* provides a set of hooks at the language level that allow programmers to exploit and control run-time behavior of the code without having to master all of the instrumentation details.

The remainder of this paper is organized as follows. The next section presents a concrete example of the application of the *selector* concepts. Section 3 describes the design and implementation of the selector in more detail. Section 4 presents preliminary experimental results of using the *selector* in a large scientific application. We discuss related work in Section 5 and conclude in section 6.

## 2 Example

We now illustrate the application of the *selector* construct in the context of solving large sparse linear systems. In this example we wish to select between three alternative equation reordering algorithms, namely Weighted Nested Dissection (WND), Multiple-Minimum-Degree (MMD) and the Multi-Section (MS). The overall objective of any equation reordering algorithm is to minimize the number of non-zero entries that arise during factorization of the matrix. Minimizing the matrix *fill* results in lower data requirements with the subsequent reduction in number of data memory accesses and arithmetic operations. To address the uncertainty of which method performs the best for a given matrix we define a *selector* as depicted in Figure 1. The *selector* construct defines a set of entries via the `entry` keyword. The *selector* also defines a symbolic name, in this case `Solver` and a list of parameters to be used by all of the entries. In addition to the binding of the entries to a single symbolic name, the *selector* defines for each entry a masking function, a *probe* function, and a *cost* function. Typically the *probe* functions are not considered as doing any useful work in the sense that they create side effects that are non-critical to the overall computation. It is the programmer's responsibility to make sure that probe functions are side-effect-free. These *probe* functions have their arguments either drawn from the *selector* parameter list or environment variables such as `clock`. Environment variables are denoted with the modifier *env* and indicate that the corresponding variable should be sampled before and after the corresponding *probe* function executes. The *cost* functions can be defined elsewhere and need not to be defined in the scope of the *selector*. We have also added, for illustration purposes, a simple *masking* function defined by the `when` keyword. In this case the *masking* function is replaced by the simple predicate (`neq > 1024`) meaning that whenever the predicate does not hold during the evaluation of the various entries, the corresponding entry is not considered.

In this example we have also defined a *policy* function and a *trigger*. A *policy* function defines how to choose the best of the set of evaluated entries. The default *policy* function is to choose the *entry* with the minimum evaluated cost. A *trigger* function defines when should the various entries be evaluated by executing the corresponding *probe* functions. In this example we have defined a parameterized *trigger* function that is active once every `block` invocations of the *selector*. An important point to notice about these functions is that they are defined in the lexical scope of the *selector*. This allows for programmers to access a set of *selector* predefined internal variables generated automatically by the compiler. These variables include for instance the number of entries in the *selector*, (`number_entries`) or the invocation number of the *selector* (`invocation_number`). Other cost related variables are declared implicitly by the *cost* functions on an entry-by-entry basis. The *selector* uses these variables to store the cost metrics associated with each *probe* function to be used for quantitative evaluation of the *cost* functions. Figure 2 illustrates the compiler generated C code for the *selector* in Figure 1. For brevity we have omitted all of the operational code but rather focused on the *selector* syntax.

```

selector Solver(Matrix A, Vector x, Vector b, int neq, int vol) is {
  // The list of alternative code variants.
  entry FactorMMD(A, x, b)
    when (neq > 1024) //this is a simple masking function.
    with probe orderingMMD(A)
    with cost costMetric1(vol, env clock);

  entry FactorWND(A,x,b)
    with probe orderingWND(A)
    with cost costMetric1(vol, env clock);

  entry FactorMS(A,x,b)
    with probe orderingMS(A)
    with cost costMetric2(neq, vol, env clock);

  // One of many possibly policies - typically one although more
  // than one is possible for distinct call sites of the selector.
  // Default policy is to choose the variant with the lowest cost.

  int policy MinCost() {
    int i, idx, min=-1;
    // It is a run-time error if no version is selectable...
    // compiler inserts check. This is also the default policy function.
    for(i = 0; i < number_entries; i++){
      if((selectable[i] && (cost[i] < min)){
        min = cost[i];
        idx = i;
      }
    }
    for(i=0; i < number_entries; i++)
      selectable[i]=FALSE;
    return idx;
  }

  // The default trigger function is { return TRUE; }
  boolean trigger Every(int block){
    if((invocation_number % block) == 0)
      return TRUE;
    return FALSE;
  }
} // end of the selector construct.

// Invocation site with specific policy and trigger function.
// Different call sites could have different policies and
// trigger functions declared for this selector
selector Solver(args) with policy MinCost() with trigger Every(10);

```

**Fig. 1.** Example of Linear Solver with Selector Construct.

```

typedef struct Solver_selector_data {
    int number_entries, invocation_number, selected_entry;
    int selectable[3]; // to reflect the masking
    double cost[3]; // to store the results of cost functions
    // Cost variables for each entry extracted from cost function args.
    int probe0_var0; // for the vol variable
    double probe0_var1[2]; // env variable with before, after
    int probe1_var0; // for the vol variable
    double probe1_var1[2]; // env variable with before, after
    int probe2_var0; // for the neq variable
    int probe2_var1; // for the vol variable
    double probe2_var2[2]; // env variable with before, after
}
...
Solver_selector_data SolverCS0;
// The Selector call site is replaced by the function below.
// Compiler initializes the Solver_selector_data SolverCS0.
...
void SolverCallSite0(Matrix A, Vector x, Vector b,
    int neq, int vol, int trigger_arg0){
    if(Every(trigger_arg0) == TRUE){ // The trigger function invocation
        for(i = 0; i < SolverCS0.number_entries; i++){
            switch(i){
                case 0:
                    if(neq > 1024) SolverCS0.selectable[0] = TRUE;
                    SolverCS0.probe0_var1[0] = clock; // before the probe executes.
                    orderingND(A,x,b);
                    SolverCS0.probe0_var0 = vol;
                    SolverCS0.probe0_var1[1] = clock; // after the probe executes.
                    SolverCS0.cost[0] =
                        (double)costMetric1(SolverCS0.probe0_var0,SolverCS0.probe0_var1);
                    break;
                ... // other cases here.
            }
        }
        SolverCS0.selected_entry = MinCost(SolverCS0);
        if((bounds(SolverCS0.selected_entry,SolverCS0.number_entries)){
            printf(" *** Error: Empty selection (solverCS0) "); exit(1);
        }
    }
    switch(SolverCS0.selected_entry){ // Now selector the variant and execute it.
        case 0:
            FactorND(A,x,b);
            break;
        ... // other cases here.
    }
}
// The original call site would be transformed into
SolverCallSite0(A, x, b, volume, number_equations, 10);

```

**Fig. 2.** Selector Code for Selector in Figure 1.

At the *selector* call site the programmer can associate an actual *policy* and *trigger* function and/or provide specific argument values. In the example above the *selector* will reevaluate the set of available entries every 10 invocations. During the successive invocations when no evaluation is required the *selector* uses the code *entry* selected in the previous evaluation. At startup the *selector* forces an evaluation to set up initial conditions. We now describe in detail the behavior of the *selector* generated C code. As outlined in Figure 2, the compiler generates code that evaluates each of the individual probes in the *selector* and chooses the entry as dictated by the user-provided *policy* function. For each *entry* the generated code first determines if that particular *entry* is selectable. It does so by evaluating the predicates (and in general a boolean function) associated with each *entry*. Next the generated code invokes the *probe* function and evaluates its run-time performance using the corresponding *cost* function. The next step is for the generated code to invoke the *cost* functions and then its *policy* function. In a typical application the programmer would like to reassess the various alternatives from time to time or in a response to a significant event. For this purpose the *trigger* function is executed before the *selector* assesses the various code variants. If the *trigger* function is inactive, the *selector* chooses the code variant selected in the previous invocation for the same call site or forces the evaluation of the probes if this is the first time the *selector* is invoked. This example illustrates the scenarios the *selector* is designed to handle. First there is a discrete number of alternative implementation for the same functionality. Associated with each *entry* the programmer can define a specific *cost* function and a set of identifiable variables whose run-time values are needed to assess the cost of the *entry*. Last the programmer can define a *policy* and *trigger* functions to control when the choice of code variants should be reevaluated and which variant should be selected. In the next section we describe in more detail the implementation issues of the *selector*. We also describe a set of more advanced features for the *selector*, which include the ability to terminate a sequence of evaluation of *probe* functions as well as the ability to abort the evaluation of a *probe* function based on a time-out specification.

### 3 Selector Design and Implementation

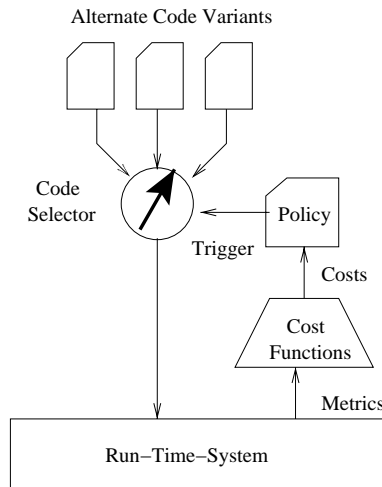
We now describe the basic concepts the *selector* relies on, their syntax and implementation restrictions. Later we describe a series of advanced features that allow for greater flexibility in specifying a richer set of *selector* behaviors for applications that require more sophisticated dynamic behavior.

#### 3.1 Basic Concepts

The *selector* fundamentally relies on four basic concepts illustrated in Figure 3, namely:

- A discrete set of alternative code variants or implementations. These variants must draw their input arguments from a common parameter list to ensure that the arguments passed at the call site to the *selector* can be applied to any of the code variant of the *selector*;

- A set of *cost* functions one per alternative code variant. These *cost* functions provide two pieces of information. First a way to generate a quantitative metric that can be used to rank code variants. Typically *cost* functions will yield floating-point values for comparison purposes. The second piece of information is that the argument list of each cost function implicitly defines the set of metric variables (*e.g.*, wall-clock or number of cache misses) to be used in the evaluation of each code variant. The compiler instruments each code variant based on these variables.
- A *policy* function that defines which of the code variant to choose. The default *policy* function, should it be omitted at the *selector* call site, is to choose the code variant with the minimum cost.
- A *trigger* function that dictates when the *selector* should evaluate the alternative code variants. This function is used to control the amount of time devoted to evaluate alternative code variants mitigating any substantial performance overhead in the search for the best code variant. The default behavior is to evaluate every code variant only once during the first invocation of the *selector*. This situation occurs when the relative performance of the code variants does not change over time but is unknown at compile time.



**Fig. 3.** Graphical Illustration of the Selector Concepts

### 3.2 Selector Syntax and Semantics

In terms of the syntax of the *selector* we have chosen a syntax that is closely related to C. The definition of the *selector* is accomplished by the `selector` construct as outlined below.

```

selector Name (type parm0,...,type parmN) is {
  entry entryName0(arg0,...,argk)
  with cost Cost0(type carg0,...,type cargC);
  ...
  int policy PolicyA(type parg0,...,type pargP) { ... }
  boolean trigger TriggerT(type targ0,...,targT) { ... }
}

```

Here the parameter list defines the names inside the scope of the *selector* that can be used for the binding of parameters for each entry and the lists of arguments `carg0, . . . , cargC`, `type parg0, . . . , type pargP` and `type targ0, . . . , targT` are simple variable expressions drawn from the *selector* parameters or globally visible variables. At the *selector* call site the programmer must include the keywords `probed selector` or the keywords `sampled selector` with an argument list that agrees in number and type with the argument list declared in the *selector*. In the *selector* definition the programmer can specify a series of *policy* and *trigger* functions. At the call site the programmer indicates which of the defined *policy* and *trigger* functions that particular call site should use as illustrated below.<sup>1</sup>

```

probed selector Name(args) with policy P(args) with trigger T(args);
sampled selector Name(args) with policy P(args) with trigger T(args);

```

We implement two distinct behaviors for the *selector*, namely a *sampled* behavior and a *probed* behavior. The *probed* selector corresponds to the implementation described in section 2. In this behavior, the implementation evaluates all of the probes of the selectable variants when triggered by an active *trigger* function. The *probe* functions are executed in turn along with their *cost* functions. Next the implementation selects a code variant according to the policy function and executes it. A *sampled selector* does not use any *probe* function for the evaluation of each code variant, but rather the code variant itself. In this situation, and because the actual code variant produces useful work, the implementation cannot simply discard the work. As such a *sampled selector* will work across *selector* invocations and use the actual run of the code variant to determine the code variant cost. At the first invocation the *selector* determines which code variants are enabled. It invokes the first of such code variants and saves the resulting performance metrics as specified by the arguments of the *cost* functions. The implementation then tracks which code variant is to be sampled in the subsequent invocations using internal variables.<sup>2</sup> At each *selector* invocation the compiler generated code first checks if the trigger function holds. If that is the case the selector enters a sampling phase where every subsequent call to the *selector* are used to sample the performance of one of the selectable code variants. Once all of the code variants have been examined during a sampling phase the selector

---

<sup>1</sup> In many cases the keyword `probed` can be omitted as long as for every entry there is a probed function. The `sampled` keyword, however, must be included.

<sup>2</sup> The *sampled selector* implementation evaluates the predicates that dictate which variants are selectable, the predicates when the *trigger* function is first active and during the next N-1 invocations, when the *selector* is sampling the various variants the corresponding *trigger* functions could possibly no longer be active.

uses the *policy* function to select which code variant to execute. Figure 4 outlines the code generation scheme for the *sampled selector* behavior.

```

// additional selector control variables
int current_entry; // the next code variant to be evaluated
boolean sampling_phase; // controls the sampling across invocations
...
void NameSelector(parm_list){
  if(triggerfunction(arg_list)){
    sampling_phase = TRUE;
    current_entry = 0;
  }
  if(sampling_phase){
    current_entry = nextEntry(current_entry,N,order,selectable);
    switch(current_version){
      ... // invoke the code variant here.
    }
    if(current_entry == number_entries){
      sampling_phase = FALSE;
      selected_entry = policy(arg_list);
    }
  } else {
    switch(selected_entry){
      ... // invoke the selected version in non-sampling phase mode.
    }
  }
}

```

**Fig. 4.** Sampled selector Code Generation Scheme Outline.

### 3.3 Advanced Selector Features

We now describe a set of advanced *selector* features namely, state and environment variables an early cut-off function.

**Selector State Variables** Another aspect of the *selector* is the ability to define **state** variables the programmer can use to define a richer set of policies using *selector* invocation site history. These state variables are declared as ordinary variables in the scope of the *selector* and therefore used by any function defined in the same scope. Eventually this allows the programmer to control aspects such as the evaluation order of the code variants and consequently ameliorating potential run-time overheads of the *selector*.

**Environment Variables** Typically a *probe* function would require a given environment variable such as wall-clock time or any other raw performance metric to be examined *before* and *after* the actual *probe* function executes. To address this,

we define the *env* attribute for variables to be used as arguments in cost functions. and associated two predefined functions `before` and `after` to access the value of the variable before and after the *probe* function executes. The compiler automatically instruments the *probe* function to extract and store at run-time the values of the environment variables before and after the *probe* function executes.

**Early Cut-Off or Break Function** In some cases the programmer might want to exploit properties of the various code variants in a given selector or simply take advantage of the information gathered in previous evaluations to terminate the evaluation in the current evaluation cycle. At least two scenarios are likely:

- The last evaluated function has yielded good enough expected performance result and no resources should be devoted to exploring alternative variants;
- The last evaluated version exhibits poor performance and subsequent variants are likely to exhibit worse performance due to monotonic properties of the implementations.

To address these concerns the selector includes the possibility of defining an early cut-off, or *break* function for each selector entry as outlined below.

```
entry E
  with cost C() with probe P() with break B(...)
```

After the *selector* has evaluated a given entry's *cost* function it invokes the corresponding *break* function. If this *break* function evaluates to the `boolean true` value the *selector* skips the evaluation of the remaining entries and selects the best implementation evaluated so far.

### 3.4 Discussion

One of the guiding principles behind the definition of a *selector* and its syntax was to keep it C-like. The intended target programmers will be a knowledgeable programmer intending to tune a library code or the sophisticated programmer whose need for performance warrants the exploitation of algorithmic trade-offs without engaging in low-level run-time environment programming. The choice of a new script-like language that would include concepts such as "best" and "cost" would present the challenge of teaching yet another scripting language to a programmer whose native programming language will be almost likely C/C++. As such we have define a set of modest extensions to C/C++ to provide hooks so that programmer can explore the notions of dynamically choosing between multiple implementations while retaining a clear, straight-forward vision of that the semantics of the *selector* is. While the approach outlined above can be viewed as fancy C++ template building there are several differences. First the *selector* compiler can perform a substantially more flexible code generation that is currently possible with templates (or at least for currently stable compiler implementations). Furthermore the type checking in the *selector*, while less sophisticated provides for a clearer semantics than in C++ using inheritance rules in templates. Second the *selector* concept is language-independent. Overall our approach has been to offer a simple set of abstractions via a modest set of new keywords and constructs.

## 4 Experimental Results

In this section we describe a preliminary experiment of the application of the *selector* concept described above to a real engineering code.

### 4.1 Example Application and Methodology

For this experiment we have used an existing FORTRAN 77 code for solving a large linear system of equations using a variant of the Cholesky direct factorization method. The segment of the code is structured into three main phases. In the first phase the code reads the input matrix from a file. In the second phase the code factors the input matrix into a lower and upper triangular matrices. In the last phase the code solves the system of equations using two back solve steps. Currently this code can use in isolation one of two competing methods for equation reordering, namely the Multiple-Minimum-Degree (MMD) [9] and the weighted nested dissection (WND) [8] ordering.

Using in this linear solver application we have coded a selector with two entries for the MMD and the WND ordering methods. We also use a cost function that uses the number of non-zero entries in the symbolic-factorization of the matrices as a prediction on how well the corresponding ordering will perform during factorization. At the time of this writing our front-end parser and code generator are not fully operational. As such we have generated the selector code manually using the template that parser will eventually use. This approach allows us to develop a sense for the implementation and performance evaluation details before investing a whole set of resources to a fully automated implementation. The original version of this solver code was written in FORTRAN 77. We have converted the driver component of this application to C to integrate with the selector generated C code. For these experiments we have used 3 input matrices referred to respectively as the **Hood**, the **Knee** and the **I-Beam**. Table 1 summarizes the relevant structural and numeric characteristics of each matrix.

**Table 1.** Input Data Sets of Linear Solver.

Application	Input Domain	Structure (if any)	Number of Equations	Storage (Mbytes)
<b>Hood</b>	Automobile Metallic Structural Analysis	2D 6-point stencil	235,962	200
<b>Knee</b>	Human Prosthetic Implant	3D localized neighbors FEM	69,502	553
<b>I-Beam</b>	Civil Engineering	regular linear	615,600	1,311

### 4.2 Results

We now describe the performance experimental results obtained with the Solver selector example described above for each of the input matrices. These experiments were carried out on a Sun Blade 100 Workstation with 1Gbyte of internal RAM and running the Solaris 8 operating system. All codes, both FORTRAN and C were compiled with the SunPro compiler using -O optimization level.

We begin this discussion by presenting in Table 2 for each input matrix the time breakdown for solving one linear system with each of the available ordering methods for a single solving run. In reality the applications are structured as multiple solve operations for the same symbolic factorization steps. Table 2

**Table 2.** Execution time Breakdown for 3 Input Matrices and 2 Ordering Methods.

Application	Step	MMD		WND	
		Time (secs)	Percent	Time (secs)	Percent
<b>Hood</b>	Ordering	4.49	3.26	8.04	7.81
	Factorization	130.79	94.85	92.55	89.94
	Solve	2.59	1.73	2.31	2.24
	Total	137.87	100.0	102.90	100.0
<b>Knee</b>	Ordering	2.93	0.97	7.05	5.23
	Factorization	296.11	98.15	126.04	93.48
	Solve	2.66	0.88	1.74	1.29
	Total	301.70	100.0	134.83	100.0
<b>I-Beam</b>	Ordering	8.96	1.03	19.25	3.40
	Factorization	731.98	84.38	418.11	73.91
	Solve	126.49	14.49	128.33	22.69
	Total	867.87	100.0	565.69	100.0

reveals that overall the symbolic factorization has a fairly small weight in the total execution time. However, the choice of which ordering, has a substantial impact on the overall factorization and consequently the overall execution time. Another observation is that although MMD ordering is usually the fastest it produces for these examples the worst execution time. For long runs of solve steps the extra computation power devoted to WND yields substantial gains, even for a single solve step. Next we report on the utilization of a probed selector with the two distinct orderings as referred above. For each solving variant we have an ordering function as its probe and use the number of non-zeros in the symbolic factorization obtained during the ordering as the prediction of the performance of the factorization step. We use the default policy, as the policy that selects the code variant with the lowest cost metric. For this experiment we enabled all the available code variants and therefore execute all of the associated probes. To evaluate the performance impact of the selector we experience with three selector strategies using trigger and break functions. For this experiment we perform 10 consecutive factorizations of matrices with the same structure followed by a corresponding solve phase. This reflects a computation where the matrix values change slightly but retains the same connectivity.

- *Probe-at-Start*: In this strategy the *selector* probes all of the available variants at the beginning of the execution and then uses the best code variant throughout the remainder of the computation.
- *Check-in-Middle*: In this strategy the *selector* probes all of the code variants at the beginning of the execution. In the middle of the execution the selector reevaluates all of the variants.

- *Reorder-and-Break*: In this strategy the *selector* reevaluates the code variants but starts with the variant that was last selected for execution. In addition it skips the remainder probes if the newly evaluated cost is either better or at most 10% worse than the cost last evaluated for this variant.

Table 3 presents the execution breakdowns for each of the input matrices and for each of the strategies described above. We have separated the amount of time devoted to the probes and compared it with the total execution time. The

**Table 3.** Execution Results for Various Selector Strategies.

Application	Strategy					
	Probe-at-Start		Check-in-Middle		Reorder-and-Break	
	Total	Selector (%)	Total	Selector (%)	Total	Selector (%)
<b>Hood</b>	1048.71	12.46(1.3)	1059.43	25.27(2.5)	1052.98	(2.1)
<b>Knee</b>	1339.25	9.71(0.8)	1349.44	19.46(1.6)	1332.70	12.60(1.3)
<b>I-Beam</b>	8699.10	28.26(0.32)	8779.83	56.35(0.6)	8864.88	36.67(0.41)

results in Table 3 reveal that the overhead associated with the probe functions is very small. Even in the case of the naive *Check-in-Middle* strategy the overhead climbs only to a modest 2.5%. This is partly true given the fact that we had only 2 competing variants of code. For larger number of variants the overhead can quickly become significant. As expected the strategy *Reorder-and-Break* reduces the overhead and could be a strategy of choice for selector with large number of code variants. Another noteworthy aspect is that the selector is only as good as the probe functions are. In the case of the **I-Beam** matrix the cost function guides the selector to choose the MMD ordering. This turns out to be the wrong choice for the factorization. A way to combat the inherent unreliability of estimations/predictions of the probes is to use the *sampled selector* version where the real factorization would be used for evaluation.

### 4.3 Discussion

This example illustrate the potential of the selector construct as a way to relieve the programmer of having to deal with explicitly building the instrumentation, control for evaluation and all of the auxiliary data structure associated with a selection scheme. The low programmer effort required to control the overall strategy of the selector is a key aspect of the overall approach. In these experiments we had to code only 20 lines of C code to specify both the *trigger*, *cost* and *break* functions. The generated C code was about 200 lines long.

An aspect not focused so far on parallel execution. Clearly parallel execution can exacerbate program execution trade-offs and hence make the application of the selector construct even more appealing (see [6] for an illustrative example). Another important aspect is that parallel execution enables the concurrent evaluation of alternative code variants thereby mitigating the selector overhead even further. We are currently working on extending the code generation scheme to support parallel evaluation as well as testing the abort functionality.

## 5 Related Work

We begin by describing these efforts and then describe the more automated compiler approaches that rely on dynamic profiling to validate or enhance the applicability of traditional compiler analysis or transformations.

### 5.1 Languages for Adaptivity

Voss and Eigenman [10] defined a new language, **AL**, used to describe compiler optimization strategies. This language uses specific constructs such as `constraint` and `apply-spec` to interface with internal compiler transformation passes and focuses on the application of compiler transformation to regions of the code that meet these constraints. The specification also allows the definition of phases of optimization and user-defined strategies. Our approach is very similar to **AL** in terms of the concept of observing run-time performance and choosing the "best" code variant. A major difference is that **AL** focuses on tuning the application of compiler transformations whereas we have focused on user-level application tuning. Adve, Lam and Ensink [1] proposed an extension to the class hierarchy of an object-oriented model of computation with three basic concepts - adaptors, metrics and events. The notion of selector described here is very object-oriented as well. Rather than relying on the syntax and class mechanisms of C++ we have chosen to use a more language-independent approach by requiring the programmer to specify the same set of concepts via functions.

### 5.2 Dynamic Compilation

Dynamic compilation aims at delaying the entry compilation process until run-time. One of the benefits is that run-time data values are accessible and specialized versions of the code can be generated on-the-fly overcoming the inherent limitations of static optimizations. While this approach has been successful in the context of interpreted languages (*e.g.*, in the Hot-Spot compiler from Sun Microsystems) for C the overheads of implementation seems prohibitively high [4, 7], thereby limiting the widespread adoption of this technique.

### 5.3 Feedback-Directed Optimization

In the context of dynamic feedback optimizations researchers have developed fully automated systems that are capable of using run-time profiling information to validate or simply apply transformations. Agesen and Hölzle use run-time information to improve the efficiency of the Polymorphic-In-Line cache (PIC) via the reordering the sequences of the tests to perform faster dynamic dispatching. Bala *et. al* [5] use run-time basic-block statistical data to reoptimize sequences of basic blocks, recompiling non-trivial sequences of basic blocks. The Jalapeño RVM project [3] at IBM uses a simple cost/benefit analysis at run-time to determine which level of optimization to apply when recompiling hot methods. Whereas previous approaches have focused on using run-time data to either validate or enhance the applicability of a given set of transformations, in our own work we have focused on using dynamic feedback to delay the binding of a different set of code variants produced with distinct policies of the same set of transformations [6]. In this approach the compiler, and therefore the compiler writer, must be fully aware of an inherent trade-off for the applications of a given set of programming transformations with distinct policies.

## 6 Conclusion

In this paper we have presented a set of modest extensions to C to allow programmers to easily specify and control the invocation of various code variants for the same functionality. The compiler can generate code automatically using a simple template relieving the programmer from the tedious and error-prone low-level performance evaluation. We have validated this approach for a sophisticated engineering linear solver. For this code the programmer can easily specify a set of cost and run-time strategy functions so that the resulting generated selector code can choose the best available code implementation while exhibiting negligible overheads.

**Acknowledgements:** We acknowledge the help of Dr. Robert Lucas with porting and explaining the original MMD and WMD. We also acknowledge the funding of this work by DARPA under contract #F33615-01-1-1890.

## References

1. V. Adve, V. Lam and B. Ensink, Language and Compiler Support for Adaptive Distributed Applications. In Proc. of the *ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)* Snowbird, Utah, June 2001.
2. O. Agesen and U. Hölzle, Type Feedback vs. Concrete Type Analysis: A Comparison of Optimization Techniques for Object-Oriented Languages. In Proc. of the *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)* 1995.
3. M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney, Adaptive Optimization in the Jalapeño JVM: The Controller's Analytical Model. In Proc. of the *ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec., 2000.
4. J. Auslander, M. Philipose, C. Chambers, S.J. Eggers and B.N. Bershad, Fast, Effective Dynamic Compilation. In Proc. of the *ACM Conference on Programming Language Design and Implementation (PLDI96)*, May 1996.
5. V. Bala, E. Duesterwald, S. Banerjia, Dynamo: A Transparent Dynamic Optimization System. In Proc. of the *ACM Conference on Programming Language and Implementation (PLDI00)*, June 2000.
6. P. Diniz and M. Rinard, Dynamic Feedback: An Effective technique for Adaptive Computing. In Proc. of the *ACM Conference on Programming Language Design and Implementation (PLDI97)*, June 1997.
7. D. Engler, Vcode: a retargetable, extensible, very fast dynamic code generation system. In Proc. of the *ACM Conference on Programming Language and Implementation (PLDI96)*, June 1996.
8. G. Karypis and V. Kumar, Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96-129, Jan. 1998.
9. J. Liu, Modification of the minimum degree algorithm by multiple elimination *ACM Transactions on Mathematical Software*, 11(2), pp. 141-153, Jun. 1985.
10. M. Voss and R. Eigenmann, High-Level Adaptive Program Optimization with ADAPT. In Proc. of the *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, 2001.