A Kernel-Based Approach for Functional Test Program Generation *

Po-Hsien Chang[1], Li-C. Wang[1], Jayanta Bhadra[2]
[1]Department of ECE, UC-Santa Barbara
[2]Freescale Semiconductor, Inc.

Abstract

This paper proposes a kernel-based functional test program generation approach for microprocessor test and verification. The fundamental idea in this approach is to select high quality test programs before the simulation from a large number of biased random test programs. Unlike a direct test program generation approach, a selection approach demands much less domain knowledge and intervention from its user for achieving a similar coverage goal, making it more applicable for scenarios targeting on different coverage objectives. We will demonstrate the effectiveness and efficiency of such an approach through performing experiments on a MIPS processor design.

1 Introduction

Automatic test generation continues to be a challenging research area. The source of such challenges lies in the complexity of search. For manufacturing testing, ATPG search complexity can be dramatically increased if statistical timing variations and noise are introduced into the search model. For the functional verification, functional test generation has to overcome the tremendous design complexity in a full-chip RTL model. The problem becomes even more challenging when tests are generated for post-silicon validation where functional tests are required to expose performance and/or marginality issues, which may depend on statistical timing variations and other types of noises.

Randomization is a common approach adopted in practice to avoid the complexity of test generation. If searching for a test to hit a target is too complex, then we can rely on randomization to produce a large number of random tests, in hope that one of the tests will hit the target.

Nevertheless, the effectiveness of a randomized test generation approach is limited by its efficiency in test application. For example, in manufacturing testing, random tests are rarely used as scan tests due to the concern of test application efficiency. Random tests are commonly used with BIST to avoid slow scan operations. In functional verification, the efficiency of a full-chip simulation limits the number of tests applied in a design cycle.

Guided random test generation (guided by user) intends to avoid the complexity of deterministic search and overcome the deficiency of pure random test generation. In manufacturing testing, weighted random test generation can be seen as a form of guided random test generation. It is commonly known that a weighted random test scheme can achieve the same coverage with much fewer tests than pure random tests. In functional verification, a popular idea for guided random test generation is constrained random verification (CRV) [1], where a user provides biases and constraints into test templates as inputs to CRV, allowing the CRV to guide its automatic generation of functional tests.

As for microprocessor verification, the idea of guided random test generation was explored in the Pseudo-random Test Program Generation (RTPG) methodology [2] originated from IBM [3]. RTPG ensures the validity of randomly generated test programs, i.e. satisfying all architectural constraints and producing meaningful results.

In CRV/RTPG, a user is required to provide *test templates* from which the functional tests are derived. A test template can be instantiated into many functional tests. This is in contrast to asking the user to write specific functional tests to hit a target. Although CRV/RTPG alleviate some of the burden from their users, in many cases writing test templates can be as difficult as writing specific tests. To overcome this difficulty, coverage-directed test generation (CDTG) was explored in [4]-[8]. CDTG techniques dynamically analyze the coverage results and automatically adapt the test generation process in order to improve the coverage. While many promising techniques had been proposed, CDTG still remains to be an on-going and active research area.

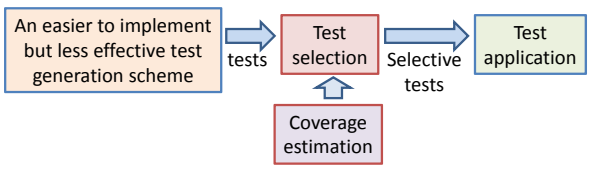

Figure 1. Illustration of test selection

*This work is supported by Semiconductor Research Corporation, Project Number 2008-TJ-1848

978-1-4244-7207-9/10/$26.0

Test selection is an alternative to test generation. If generating effective tests is too difficult, test selection can select effective tests from a large number of tests.

As illustrated in Figure 1, if implementing an effective automatic test generation scheme is too difficult to accomplish, a less effective test generation scheme would be used. A large number of tests are produced and then filtered by a test selection process. In order to produce effective tests, it needs to incorporate coverage estimation so that tests can be evaluated for their effectiveness before application.

The idea of test selection was implemented in [9] [10], in the context of testing for statistical delay defects. Since ATPG with a statistical timing model is difficult, a transition fault ATPG is used. Although a transition fault ATPG is not optimized for statistical delay defects, it is much easier (and realistic) to implement in practice than a statistical timing ATPG. For coverage estimation, a statistical timing simulator can be used [10], which is also much easier to implement than a statistical timing ATPG. Tests are then selected based on the results provided by the statistical timing simulation.

In the test selection [10], part of the ATPG problem is converted into a simulation problem. In order for the test selection approach to be practical, the simulation efficiency has to be high and the coverage estimation has to be accurate in the sense that if a test is estimated to have a higher coverage than another test, this test is also likely to capture more statistical timing defects in test application.

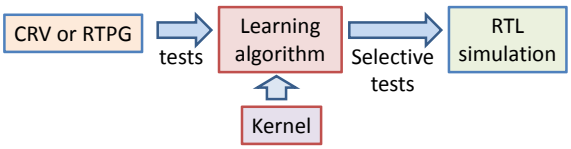

Figure 2. Kernel approach for functional test selection

In the context of functional verification, the idea of test selection was first implemented in [11]. Figure 2 illustrates its basic idea analogous to Figure 1. In this approach, practical methodologies such as CRV or RTPG are used to produce a large number of functional tests. For the coverage estimation, however, a simulation approach is not favored. This is because if a simulation approach is adopted, it implies that we need to have a simulation scheme that is orders-of-magnitude faster than the RTL simulation (the target test application); otherwise, we would lose the purpose of test selection. One possibility of achieving fast simulation is by simulating a high-level reference model, such as a behavior model. However, in many designs, such a high-level reference model does not explicitly exist.

The authors in [11] avoids the use of simulation in coverage estimation by introducing the use of *kernel* [12] in conjunction with a learning algorithm, in particular the Support Vector Machine (SVM) one class algorithm [13], to achieve the functional test selection. Note that the kernel used in this context is not the same definition widely used in computing which is the central component of the most computer operating system. It refers to the kernel in the *kernel methods* [12] which becomes part of the mainstream in the machine learning field. The work in [11] was based on a CRV environment and the test selection was designed for fixed-cycle functional tests. Results were shown on selected units in OpenSparc T1 processor to demonstrate the feasibility of the approach.

The idea of using kernel in place of coverage estimation was not fully explored in [11]. Moreover, the test selection scheme was limited to fixed-cycle functional tests. These two aspects both point to the same fundamental question for such approach: Why using a kernel, without simulation, is sufficient to achieve the required objective of coverage estimation for effective test selection?

This work is an attempt to approach this fundamental question. In this work, we consider RTPG instead of CRV in test generation. With RTPG, guided random test programs of various lengths are produced. Then, we apply the functional test selection approach in Figure 2 to select effective test programs for the purpose of functional verification measured by various coverage metrics. Because the selection is applied on test programs, special kernels are developed for analyzing test programs. Note that the kernel used in [11] can not be applied here because it can only analyze test programs in the same length. In this work, we alleviate this limitation by using a more flexible graph based kernel. Initial experimental results on a MIPS processor design are demonstrated to validate the effectiveness of the test program generation/selection approach.

The rest of the paper is organized as the following. Section 2 explains the concept of kernel fundamental to the work. Section 3 describes the functional test selection problem in detail. Section 4 discusses the development of kernels suitable for analyzing test programs. Section 5 explains the two types of learning algorithms considered in this work. Section 6 shows the experiment results and Section 7 concludes the paper.

2 Kernel in test selection

The use of kernel in place of coverage estimation is essential for the proposed test program selection approach. The fundamental concept of using kernel in test selection is the following: *For selecting tests, there is no need to know the coverage of individual tests; Instead, only information on how pairs of tests are related to each other in the coverage space is required.*

Consider we are provided with n tests, $t_1, t_2, \ldots, t_n$. Suppose we are asked to select the most effective m tests from this list to achieve a coverage objective. An intuitive thinking would be to first calculate coverage for these tests,

$C(t_1), C(t_2), \ldots, C(t_n)$. Then, based on these coverage results, a selection algorithm is applied to select the m tests of which the total coverage is maximized. As mentioned earlier, the problem of following this idea in functional test selection lies in simulation to obtain the coverage results.

While there is nothing fundamentally wrong with the above intuitive thinking, it is important to see that for test selection, calculating $C(t_i)$ for each individual test t_i may not be necessary. In other words, this "absolute information" is not required for test selection.

In test selection, most of the required information involves comparison of at least a pair of tests. This is because in a selection process, most of the time we would be asking the question of which one should be included. In other words, for the selection, "relative information" is more important and useful than "absolute information".

For example, suppose that we are given with three tests t_1, t_2, t_3. Assume we know that t_1 and t_2 are similar. t_1 and t_3 are dissimilar, so do t_2 and t_3. Suppose in a test selection process we have already selected t_1. Then, obviously for the next selection in between t_2 and t_3, t_3 should be selected. Notice that t_3 is selected without the knowledge of its estimated coverage result $C(t_3)$. All we need to know is that t_3 is more dissimilar to t_1 than t_2 is.

Instead of trying to estimate the coverage $C(t_i)$ for each individual test t_i, a kernel $k(x_i, x_j)$ estimates the *similarity* between a pair of tests. This relative information is all that is needed for a selection algorithm to work (explained in detail later in Section 4). However, in order for the test selection to work effectively, this similarity measure has to somewhat correlate to the similarity of the pair of tests in the target coverage space. Figure 3 illustrates this concept.

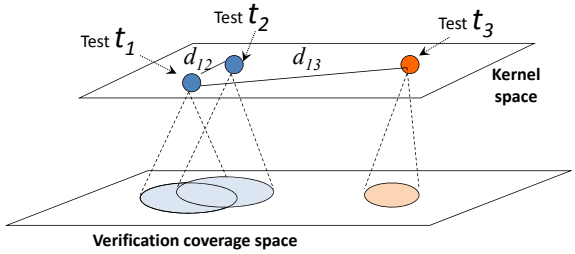

Figure 3. Kernel space vs. coverage space

The figure illustrates the "trend information" that a kernel intends to capture. In the kernel space, $d_{12} = k(t_1, t_2) < d_{13} = k(t_1, t_3)$ shows that t_1 is more similar to t_2 than t_3. For this similarity measure to be statistically meaningful, the covered area of t_1 should overlap more to the covered area of t_2 than to the covered area of t_3 with a higher probability in the verification coverage space.

A feasible kernel only solves half of the test selection problem. We still need a selection algorithm that can operate by using only the information provided by the kernel. We will postpone the discussion of the selection algorithms to Section 5. Before then, we should ask an important question: How difficult is it to develop a feasible kernel?

In a CRV or RTPG environment, a user encodes his/her knowledge about the design into test templates consisting of biases and constraints to guide the test generation process. It is unrealistic to think that developing a feasible kernel for a certain verification objective demands no domain knowledge. However, it is important to note that although some domain knowledge may still be required, developing a kernel would demand much less knowledge and would be much easier than developing test templates. Otherwise, using kernel based test selection would not be rational.

Furthermore, kernel computation has to be very efficient because the overall test selection process has to be orders-of-magnitude faster than RTL simulation.

The discussion so far shows that developing a feasible kernel is at the core of applying the kernel based test selection approach. This is why in this work, we choose test programs instead of fixed-cycle tests used in [11]. Test programs are sequential tests with much more complexity and may contain diverse instructions of various lengths. Our goal is to show that relatively simple kernels can be developed without too much domain knowledge and still enable effective selection of test programs.

It should be noted that although in this work we only consider common coverage metrics such as statement coverage, toggle coverage, branch coverage, and so on, the kernel based test selection approach can be applied to other scenarios involving different coverage objectives. While the kernels implemented in this work may be specific to test program selection for functional verification, the kernel based approach itself certainly is not.

3 Test program selection problem

Practical functional verification relies on extensive simulation. For example, in a typical processor design, billions of simulation cycles usually are spent on functional verification. Suppose during a design period, N test programs have been run to achieve a coverage goal. At the end of the period, we ask this question: If we could start all over again, how many test programs are really needed to achieve the same coverage goal?

The test selection problem is illustrated in Figure 4. Suppose the coverage metric consists of n targets are to be covered. In the coverage result table, each test covers a subset of these n targets. From this perspective, we see that selecting M tests with maximum coverage can be seen as an instance of the set covering problem, a well known NP-complete problem. In practice, such a problem is usually solved by a greedy algorithm, i.e. selecting the next test that maximizes the current coverage.

Solving the test selection problem in Figure 4 is interesting but provides little help to the on-going functional verification effort because those N test programs are already

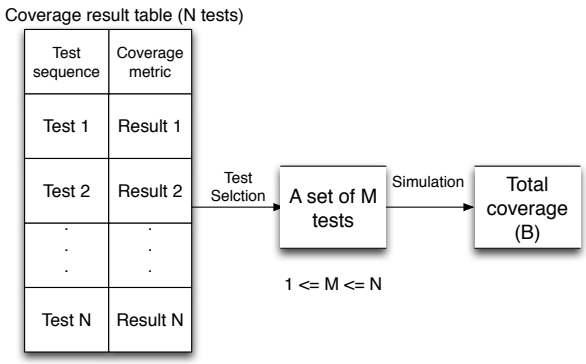

Figure 4. **Test selection after simulation**

simulated. Hence, the question should really be asked *before* the simulation. If we can determine before simulation that simulating 20% of the test programs is as effective as simulating all programs, then the simulation time will be reduced by 80%. The extra simulation time can be devoted to simulating other test programs and consequently, in the same period of time higher verification quality can be achieved. Before simulation, however, the coverage result table in Figure 4 does not exist. Hence, we cannot view the problem as simply a set covering problem.

The test program selection problem can be approached from two perspectives: on-line selection and off-line selection. In on-line selection, i test programs are already simulated. The goal is to select a small subset of j programs from $N-i$ programs to achieve maximal additional coverage. In off-line selection, all the N test programs are given. The problem is to select a (much smaller) subset of the test programs and achieve almost the same coverage objective.

In this work, we view on-line selection as a *novelty detection* problem and the off-line selection as a *clustering* problem; both are studied extensively in machine learning [12].

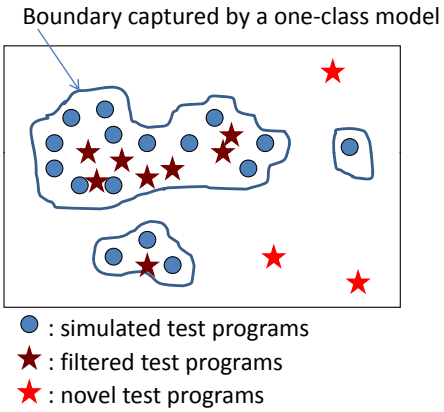

Figure 5. **Test program selection by novelty detection**

Figure 5 illustrates the selection in the on-line case. Test programs already simulated are given to a one-class learning algorithm to obtain a learning model that captures the boundary of the covered area in an estimated coverage space. The learning model is applied to measure whether a new test program is inside the boundary or not. If it is inside, the test program is filtered out. If it is outside, a distance is calculated for the test program to measure how close it is to the boundary. Test programs outside the boundary are considered as novel tests. The degree of novelty is measured by the distance to the boundary. These distances can then be used to select the j tests that are most novel.

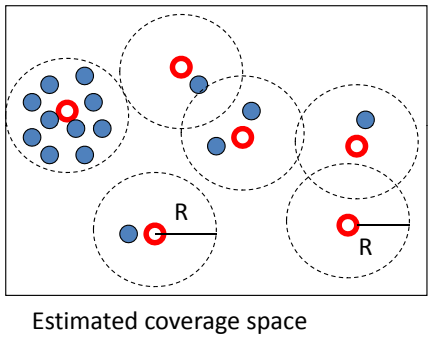

Figure 6. **Test program selection by clustering**

Figure 6 illustrates test program selection by clustering. Suppose test programs can be mapped to an estimated coverage space as points (i.e. through a kernel function). In this space, a clustering algorithm can be applied to group test programs. Test programs in each cluster are more similar to each other than to test programs in other clusters. Suppose six clusters are found. In this case, six representative test programs are selected from each cluster to maximize the coverage of the space.

4 Kernel based learning in test selection

For a clustering algorithm or a one-class learning algorithm to work, a kernel is required to defined the estimated coverage space shown in Figure 6 and Figure 5 above. Figure 7 illustrates how a kernel based learning algorithm works conceptually [12].

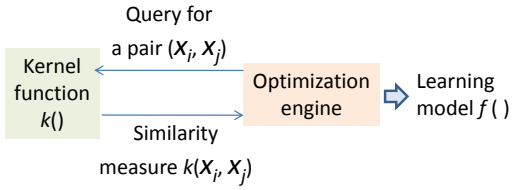

Figure 7. **Illustration of kernel-based test selection**

A kernel based learning algorithm consists of two components: a kernel function $k()$ and an optimization engine. The kernel function calculates a similarity measure $k(x_i,x_j)$ for any pair of tests x_i,x_j. The optimization engine computes the learning model. The important thing to note is that in the computation, the optimization engine works with the kernel function by sending queries for similarity measures. This is the only type of information required for it to

perform the optimization, i.e. it never accesses any information on individual test programs.

The separation between kernel function and optimization engine in kernel based learning make it a powerful approach to be applied in diverse areas [12]. For example, the SVM one-class algorithm, ν-SVM [13], is a particular optimization algorithm used for computing a one-class model. The ν-SVM optimization algorithm is independent of the kernel. Hence, one can design different kernels to be used with the ν-SVM optimization engine in different applications.

Figure 7 can be interpreted from another perspective. The optimization engine is responsible for computation in the learning process. The kernel function, on the other hand, captures a person's intuition about the learning problem. This intuition is encoded in the similarity measure. In other words, kernel function is where a user can input domain knowledge to direct the learning process.

4.1 Domain knowledge

As mentioned before, a key premise for the kernel based test selection to be practical is that developing a feasible kernel function requires much less domain knowledge from its user than developing the tests (or test templates). We use a simple example to illustrate why this is possible.

Test sequence	Operation	Operand	Operand
1	ADD	A	B
2	ADD	C	D
3	MPY	C	D

Figure 8. An example of three tests on ALU

Figure 8 shows such an example with three tests. These three tests are to be applied on the ALU block in a microprocessor. Each test consists of one type of two-operand instruction. Suppose we are not aware of that each row represents an instruction. By comparing the three symbolic vectors v_1 =(ADD,A,B), v_2 =(ADD,C,D), and v_3 =(MPY,C,D), an intuitive kernel function is that v_2, v_3 are more similar than v_1, v_2 because v_2, v_3 share two symbols while v_1, v_2 shares only one symbol. If we let the learning engine to treat the three instructions in this way, i.e. simply as three symbolic vectors, then after simulating instruction 1, the learning will select instruction 2 as the next test.

The symbolic vector view takes no domain knowledge into account. With domain knowledge, we know that ADD and MPY are instructions on ALU. For coverage purpose, instruction differences should play a more important role than operand differences. With this knowledge, a kernel should be defined so that v_1, v_2 is more similar than v_2, v_3. This kernel will then guide the test selection to select instruction 3 as the next test after simulating instruction 1.

The above example shows that without any domain knowledge, it can be unrealistic to expect the kernel based test selection to work effectively in selecting effective test programs. However, with very limited domain knowledge (i.e. just knowing the definition of an instruction set), a reasonable kernel can be developed and the knowledge about the instruction set can be incorporated into the computation of the learning model by the optimization engine.

It is important to note that the learning model illustrated in Figure 7 does not solely depend on the kernel. The optimization engine plays a key role as well. Hence, while domain knowledge encoded in the kernel can influence the learning process, the learning model is really a result from the combination of both the kernel and the optimization engine. This means that we do not need to have a "perfect" kernel for us to start using the test selection. We can start with a reasonable kernel and refine it progressively.

4.2 Graph based kernel for test programs

In this work, we develop a graph based kernel to measure similarity between two assembly programs. An assembly program is converted into a directed graph. In this graph based representation, a vertex represents an instruction, annotated by several attributes such as type of the instruction, opcode, the register usage, and the number of operands.

The data dependencies of instructions can also be captured by annotations. For example, a vertex A could be annotated to record that an earlier instruction B updates the register R1 read by A. For data dependency annotation, an instruction is annotated with its closest previous dependent instruction based on register or memory address.

Edges in a graph capture the actual execution flow of the assembly program. The number of outgoing edges of vertices is not restricted to one. For instance, the vertices that correspond to jump and branch instructions have two outgoing edges, which depend on the condition expression. Figure 9 shows an example of an assembly program, represented as a graph.

Address	Instruction	
214	addiu	a1,v0,2483
218	j	0x234
21c	addiu	v1,v1,2475
220	addu	v0,v0,a2
224	sb	v0,0(a1)
228	addiu	a1,a1,-1
22c	beq	v1,a1,0x238
230	srl	a0,a0,0x4
234	beqz	v0,0x220
238	slti	v0,a2,10

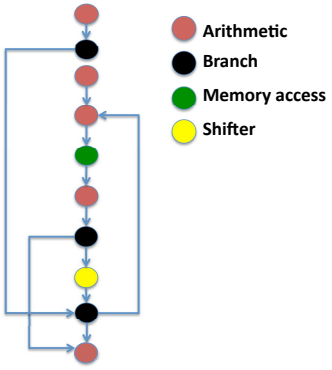

Figure 9. Assembly program in a graph representation

Let $\mathcal{G}$ be the space of all possible program graphs. Let $\mathcal{R}$ be the space of real numbers. A graph kernel function is a mapping $k : \mathcal{G} \times \mathcal{G} \rightarrow R$. In our work, the kernel $k()$ takes two program graphs G_i, G_j and outputs a numerical value $k(G_i, G_j)$ to quantify their similarity that is calculated based on a *dissimilarity* measure described below.

The dissimilarity is measured based on the difference between two graphs. A popular concept for this is called *graph edit distance* (GED). GED defines the dissimilarity of two graphs by the minimum amount of *editing* required to transform one graph into another. The edit operations include, insertion, deletion and substitution of edges and vertices. A sequence of edit operations $(e_1, \ldots, e_k)$ that transform a graph G_i into a graph G_j is called an *edit path*. In order to measure the dissimilarity, a cost is defined for each operation e_h. The total cost of the edit path is the sum of all operation costs. The dissimilarity between the two graphs is defined as the minimum cost edit path between the two.

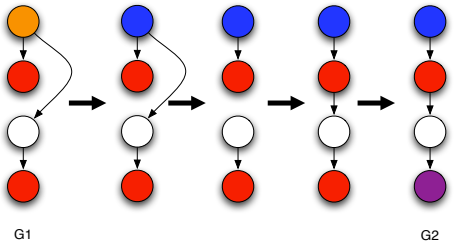

Figure 10. **A graph edit path between G_1 and G_2**

Figure 10 shows an example of an edit path between two graphs. This edit path follows the path: vertex substitution, edge deletion, edge insertion, and another vertex substitution. The dissimilarity is measured by summing the costs of these four edit operations.

There are several advantages to use graph edit distance as a similarity measure. It can capture the partial similarities between graphs. Moreover, it provides a flexible way to define costs for various edit operations, where domain knowledge can be incorporated. For example, substituting an ADD instruction with another ADD instruction has a lower cost than substituting an ADD instruction with MPY. This means that all we need to do for integrating the domain knowledge into test selection is by defining the costs associated with all possible edit operations. Since this cost definition table does not affect the graph comparison algorithm, it can be easily expanded and refined without modifying the test selection software.

Once a cost table is given, the GED can be computed by a branch-and-bound like search algorithm, where possible edit paths are iteratively explored, thus the minimum-cost edit path can be retrieved from the search tree. This method allows us to find the optimal edit path between two graphs. However, such a strategy may result in an exponential complexity, making it applicable to small graphs. Therefore, the brute-force search algorithm is not practical.

Several suboptimal method [16, 17] were proposed to overcome the complexity issue. The idea in [16] is to decompose graphs into sets of subgraphs to reduce the complexity of the problem. It tries to find an optimal matching between the sets of subgraphs by dynamic programming.

In this paper, we employ another approach proposed in [17]. It approximates graph edit distance by finding an optimal matching between nodes and local graph structure. Instead of dynamic programming, the computation is based on bipartite graph matching using Munkres algorithm [18] as a heuristic. In the worst case scenario, the complexity of the algorithm is $O((n+m)^3)$ where n, m are the numbers of vertices in the two graphs.

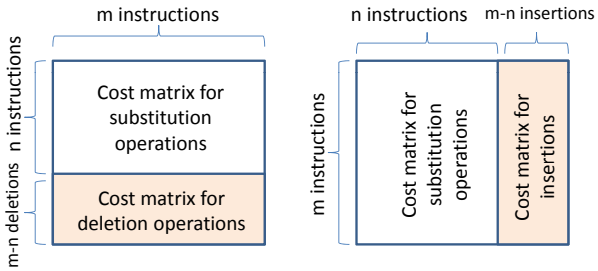

Figure 11. **Cost matrix for mapping n instructions into m instructions and vice versa, for $m > n$**

Figure 11 illustrates the basic concept in the bipartite matching heuristic. Suppose we are transforming a graph with n vertices into a graph with m vertices. The left side of the figure illustrates this situation. Each of the n vertices corresponds to a row in the cost matrix. Each of the m vertices corresponds to a column. Suppose $m > n$. Hence, $m - n$ deletions are required, which could happen on any one of the m instructions. Therefore, $m - n$ rows are added to define the costs for those deletion operations.

Given such a cost matrix, an edit path consists of selected entries in the matrix, covering each row with each column. The GED is therefore the minimum cost edit path based on the given cost matrix.

Conversely, suppose we are transforming the graph with m vertices into the graph with n vertices. The right side of the figure illustrates this situation. In this case, $m - n$ insertions are required to expand the n instructions. A cost matrix for these insertions is therefore defined.

Entries in a cost matrix are defined based on domain knowledge and utilization of recorded information in vertex annotations to estimate the cost for each operation. As described above, these annotations describe the structure of an instruction as well as its dependency to other instructions. Hence, our intuition on how different types of annotated information may influence coverage can be incorporated into the cost definition. It is important to note that this intuition is local to each operation individually. In defining the costs, we do not concern global coverage impact by a sequence of

operations because that is exactly what the bipartite matching intends to capture by finding a minimal edit path.

4.3 A kernel of kernel

Suppose we are given with N assembly programs, to obtain N program graphs $G_1, G_2, \ldots, G_N$. For each graph G_i, the graph kernel described above computes a similarity measure $k(G_i, G_j)$ for all j, $1 \leq j \leq N$. The vector $V_i = (k(G_1, G_i), k(G_2, G_i), \ldots, k(G_N, G_i))$ records how similar G_i is to all other graphs (illustrated as a symmetric matrix in Figure 12 below). Given such two vectors V_i, V_j for graphs G_i, G_j, we can then apply common kernel functions such as dot-product kernel or Gaussian kernel [12] to compute the similarity between the two vectors. For example, with the Gaussian kernel, $\mathcal{K}(G_i, G_j) = e^{-g\|V_i - V_j\|^2}$ where $\|V_i - V_j\|^2 = \sum_{h=1}^{N}(K_{i,h} - K_{j,h})^2$ and g is a parameter that decides the Gaussian width that scales the similarity measure. We call $\mathcal{K}()$ a *kernel of kernel* because it is defined based on applying another kernel function on the results from the graph kernel. Note that the kernel of kernel $\mathcal{K}()$ measures similarity for a program graph based on a global view (i.e. all program graphs), rather than a local view (only the graph to be compared).

$$
\begin{array}{c}
\begin{array}{cccccc} & t_1 & t_2 & t_3 & t_4 & t_5 \end{array} \\
\begin{array}{c} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{array}
\begin{bmatrix} 1 & \mathbf{K_{2,1}} & \cdots & \cdots & \mathbf{K_{n,1}} \\ \mathbf{K_{2,1}} & 1 & \cdots & \cdots & \mathbf{K_{n,2}} \\ \vdots & \vdots & 1 & & \vdots \\ \vdots & \vdots & & 1 & \vdots \\ \mathbf{K_{n,1}} & \mathbf{K_{n,2}} & \cdots & \cdots & 1 \end{bmatrix}
\begin{array}{c} = V_1 \\ = V_2 \\ \vdots \\ \vdots \\ = V_n \end{array}
\end{array}
$$

Figure 12. A kernel matrix

5 Novelty detection and clustering

In this work, we consider two algorithms for test selection, ν-SVM [13] for novelty detection and kernel based k-mean clustering [14] for clustering.

5.1 Novelty detection

Novelty detection is used to identify special samples in a sample set where the special samples are numerically distant from the trend established by the rest of the samples. In a two-dimensional space, novelty detection can be done easily by finding outliers. In a high-dimensional space, boundary of the trend and outliers are found by an optimization engine such as ν-SVM.

Suppose program graphs $G_1, \ldots, G_m$ correspond to the m programs that have been simulated. Let $k()$ be the kernel function. In novelty detection, we need to compute a model $S()$ to capture the space covered by the m programs. Suppose we are then given a new set of programs $P_1, \ldots, P_n$. Then, $S(P_i)$ returns a value telling if P_i is a novel program or not, i.e. whether it is in the covered space or not. Typically, if $S(P_i) \geq 0$, it means P_i is within the covered space. If $S(P_i) < 0$, the larger the absolute value $|S(P_i)|$ is, the further P_i is from the covered space.

In ν-SVM, the model $S()$ takes the following form (for a program P):

$$S(P) = R^2 - \sum_{\forall i,j} \alpha_i \alpha_j k(G_i, G_j) + 2\sum_{\forall i} \alpha_i k(G_i, P) - k(P, P)$$

For each G_i, an α_i is computed. If $|\alpha_i| > 0$, the G_i is called a support vector (SV). Otherwise, it is called a non-support vectors for the model, i.e. non-SVs have no effect in the model.

The term $\sum_{\forall i,j} \alpha_i \alpha_j k(G_i, G_j) - 2\sum_{\forall i} \alpha_i k(G_i, P) + k(P, P)$ can be seen as a weighted average of square distance between P and all the support vectors. The weights are decided by the alpha's. In the kernel space, R is the size of the radius of the hypersphere (the covered space) defined by the support vectors.

$S()$ is found by solving a quadratic optimization problem using the Lagrangian method. Therefore, each α_i is nothing but the Lagrangian multiplier for G_i [13].

In ν-SVM, each Lagrangian multiplier α_i is constrained by $0 \leq \alpha_i \leq \frac{1}{\nu m}$ where ν is a user supplied number. In ν-SVM, the support vector model has the property that (1) ν is an upper bound on the fraction of outliers, (2) ν is a lower bound on the fraction of support vectors.

Because our intent is to capture the space covered by $G_1, \ldots, G_m$, we would like to learn a model that treats almost all these programs as inliers. For this reason, we usually use a very small ν. For example, we use $\nu = \frac{1}{m}$ to ensure that at most only one of $G_1, \ldots, G_m$ would be left outside the covered space captured by the support vector model.

5.2 Kernel k-means clustering

A clustering algorithm groups similar programs together to expose regularities in the set of programs. K-means clustering is a popular algorithm for clustering [14]. In k-means clustering k clusters are found by iteratively refining an initial set of k random means. Clusters are defined as the set of samples most similar (closest) to each mean. Kernel k-means clustering simply refers to using a kernel function to measure these similarities [12]. As previously explained in Figure 6, we use clustering to select k representative test programs from a given set of test programs, each representing the sub-space covered by a cluster of programs.

6 Experimental results

We conducted experiments based on a public domain 32-bit RISC processor design, the Plasma/MIPS CPU core [15]. The RTL design of the Plasma CPU is described in VHDL with about 4800 lines of code. The Plasma core executes all MIPS I(TM) user mode instructions except unaligned load and store operations due to the patent issue. Table 1 shows the major blocks of the Plasma core.

Table 1. Major blocks of Plasma

Block name	Purpose
pc_next	Program Counter Unit
mem_ctrl	Memory Controller
control	Opcode Decoder
reg_bank	Register Bank for 32, 32-bit Registers
bus_mux	BUS Multiplex Unit
alu	Arithmetic Logic Unit
shifter	Shifter Unit
mult	Multiplication and Division Unit

To demonstrate the effectiveness of the proposed kernel based test program selection approach, we generated 1000 random assembly programs to be the total test set. We built an RTPG to ensure that randomly generated MIPS assembly programs satisfy architectural constraints and produce meaningful results. We used Mentor Graphic Modelsim as our simulator to estimate coverage based on various metrics. Each random test programs was generated based on random selection from more than 50 types of instructions such as arithmetic, logic, branch, load, store and jump instructions supported by the PLASMA core. These assembly programs, which ranges from 80 to 100 instructions, share the same boot sequence used to initialize the core. The boot sequence contains 50 instructions.

Testbench is set to have clock cycle time of 50 ns and each program takes around 10 to 20 ms for the simulation. Several coverage metrics are evaluated during the simulation, including statements, branches, expressions, conditions and toggle coverage. Figure 13 shows the accumulated coverage results collected through running the total 1000 test programs.

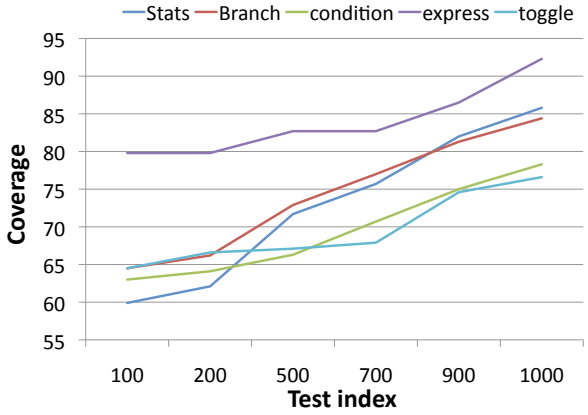

Figure 13. Accumulated coverage results

Since the boot sequence stays in the initial segment of every test program, Table 2 shows the coverage results achieved by this boot sequence, compared to the final coverage results achieved by running all test programs. The coverage results from the boot sequence should be seen as the starting point for accessing the merit of coverage contribution from each test program.

Table 2. Boot sequence vs. total coverage

Coverage metric	boot	1000 tests
Statements	57.5	85.8
Branches	60.6	84.4
Expressions	76.9	92.3
Conditions	63.0	78.3
Toggle	52.4	76.6

Table 3 shows the coverage results based on individual blocks in PLASMA after running all 1000 programs. A control block is usually more difficult to hit coverage targets. For example, the "mem_ctrl" block tends to have lower coverage numbers than others. Arithmetic block is usually the easiest. For example the "alu" and "shifter" blocks both have 100 percent achieved on all coverage metrics.

Table 3. Coverage results for the 1000 tests

Block name	Statements	Branches	Condition	Toggle
pc_next	100	91.7	100	66.5
mem_ctrl	98.4	92.9	66.7	85.2
control	84.3	84.1	33.3	95.7
reg_bank	70.6	72.2	100	97.5
bus_mux	93.1	93.1	100	90.4
alu	100	100	100	100
shifter	100	100	100	100
mult	100	100	92.9	95.6

Two types of experiments were conducted, one for the on-line selection using novelty detection, and the other for the off-line selection using clustering. In novelty detection, we first focused on the graph kernel and then employed the Gaussian kernel of kernel for comparison. In clustering, we had only considered the Gaussian kernel of kernel.

6.1 On-line test selection

The on-line test selection experimental flow is illustrated in Figure 14. In each iteration, 50 most novel test programs decided by the ν-SVM learning model are selected. These 50 test programs are then added to the selected set for learning in the next iteration. Initially, 50 test programs are randomly selected as the starting point.

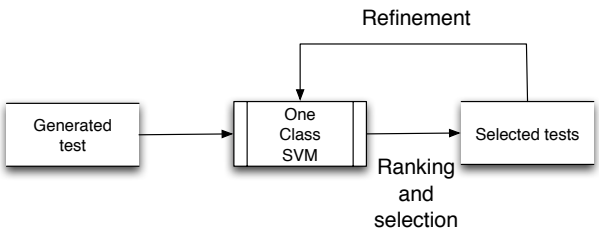

Figure 14. One-class svm iteration

In each iteration, the learning model $S()$ calculates a novelty measure for the remaining test programs. For example, in the first iteration, $S()$ calculates such measures for the remaining 950 test programs. Figure 15 shows these 950 measures in a sorted order.

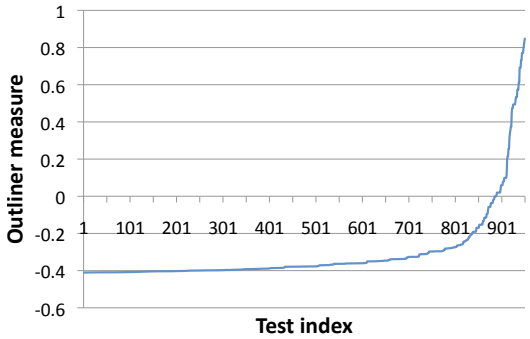

Figure 15. **Test programs ranked by novelty measure in the 1st iteration**

In Figure 15, test programs that correspond to those points above the horizontal zero line are programs that are considered by the SVM model to be inside the sub-space covered by the initial 50 programs. In other words, these programs (more than 100 such programs) are decided by SVM as those providing no "novelty" to the coverage. Others are considered to be novel programs. The degree of novelty is reflected in the novelty value computed by $S()$. The leftmost program in the plot with a novelty value roughly -0.4 is considered as the most novel program. Based on this sorted order, the top 50 most novel test programs are selected to be added into the simulated set.

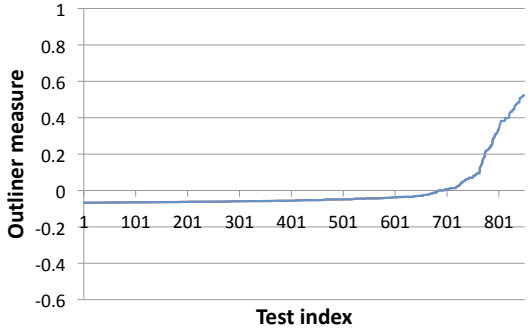

Figure 16. **Test programs ranked by novelty measure in the 3rd iteration**

Figure 16 shows the sorted novelty measure values from the 3rd iteration. In the third iteration, 150 test programs are simulated and hence, they are used to build the SVM model $S()$. The remaining 850 test programs are evaluated for selection. In Figure 16, we see that the leftmost novelty value increases to above -0.1 from -0.4 in Figure 15. This shows that after simulating 150 test programs, the most novel program in the remaining set is not as novel as that in the first iteration. This is consistent with our intuition that as more test programs are simulated, it become harder to find novel test programs in the remaining set.

Table 4 summarizes the coverage results from the first three iterations. In each iteration, we show the number of test programs that achieves the same coverage results as those obtained by running all test programs collected up to that iteration. For example, in the 1st iteration, 50 test programs are selected to make the total with 100 programs (plus the initial 50). However, simulating the 88 of them obtains the same coverage results as simulating all the 100 test programs. This shows that the selection during the 1st iteration includes 12 redundant test programs with respect to the coverage metrics evaluated in the experiment. Note that this should not be seen as a negative result because we certainly do not want the selection depends too much on the metrics in use. Hence, it is reasonable to have some selected tests to being redundant with respect to the metrics.

Table 4. **Svm iteration (graph kernel)**

Iteration	Initial	1st	2nd	3rd
# of tests	50	88	126	168
Statements	78.9	81.4	82	85.8
Branches	79.3	81.1	81.3	84.4
Expressions	86.5	86.5	86.5	92.3
Conditions	73.9	73.9	75	78.3
Toggle	74.5	74.6	74.6	76.6

It is more important to focus on the result that 168 test programs delivered the same coverage results based on all metrics as those by the entire 1000 test programs shown in Table 2. This means that on-line test selection was able to reduce the test size by 80% while maintaining the same results. Suppose the simulation time of each test program is roughly the same, then it means there is a reduction of 80% in simulation time, which in practice could be meaningful.

The 80% reduction also indicates that our graph based kernel is on the right direction. We emphasize that when defining the cost table for the graph kernel, i.e. incorporating our domain knowledge into the kernel, we did not consider any coverage metric nor analyze any part of the RTL code. However, if we were asked to write 200 test programs directly to achieve high coverage based on those metrics, we would have to look into the RTL to find holes in the coverage results and refine the programs progressively. This process would require much more domain knowledge than defining the cost table for the kernel.

Table 5. **Svm iteration (kernel of kernel)**

Iteration	Initial	1st	2nd	3rd
# of tests	50	87	132	161
Statements	78.9	83.2	84	85.8
Branches	79.3	82.9	83.4	84.4
Expressions	86.5	92.3	92.3	92.3
Conditions	73.9	77.2	77.2	78.3
Toggle	74.5	76.5	76.5	76.6

For comparison, we ran the same on-line selection experiment flow using the Gaussian kernel of kernel for comparison. The coverage results analogous to Table 4 using the graph kernel, are shown in Table 5. It is observed that

using kernel of kernel achieves better results than using the original graph kernel. As explained previously, the kernel of kernel calculates the similarity based on a global view. Hence, the better results in Table 5 are consistent with our intuition of putting more information in the kernel could deliver better result. Note that computationally, the kernel of kernel is more expensive. It is because that each graph G_i involves checking the graph similarities with all others.

6.2 Off-line test selection

The experiment was conducted in the same 1000 tests. First, we applied k-means clustering to obtain 50 clusters of test programs from the original 1000 programs. Then, a test program was randomly selected from each cluster. We compare the coverage results based on these 50 selected test programs to 50 randomly selected test programs which are the same tests as those in the initial 50 test programs used in the on-line selection experiment. Table 6 shows the comparison results. The 50 k-means programs achieved higher coverage results with respect to all metrics.

Table 6. Clustering (kernel of kernel) vs Random

Coverage metric	K-means(50)	Random(50)
Statements	82.0	78.9
Branches	82.9	79.3
Expressions	92.3	86.5
Conditions	77.2	73.9
Toggle	76.5	74.5

From Table 4, we see that 168 test programs are sufficient to deliver the same coverage result as 1000 test programs. Hence, we experimented with k-means clustering to select 168 representative programs. Table 7 shows the coverage results from these 168 representative programs, as compared to the results from the 168 programs in Table 4. We observe that the 168 representative programs achieve the same results on expressions, conditions, and toggle metrics but achieve slightly lower results on the other two.

Table 7. Clustering (kernel of kernel) vs SVM

Coverage metric	K-means(168)	Svm (168)
Statements	85.2	85.8
Branches	84.1	84.4
Expressions	92.3	92.3
Conditions	78.2	78.3
Toggle	76.6	76.6

7 Conclusion

In this work, we propose a kernel based test program selection approach to facilitate test program generation. We develop a graph based kernel to analyze test programs by capturing their similarities. In this graph kernel, domain knowledge is incorporated into a cost table that can be progressively refined without modifying the test selection software. The domain knowledge used to define a feasible cost table does not require understanding the coverage metrics nor analyzing any part of the RTL code. Experimental results show that the proposed test program selection approach was able to achieve more than 80% reduction in terms of the number of test programs required to achieve target coverage results. While the graph kernel is specific to the test program selection, the approach of kernel based test selection is general and can be applied to diverse applications as long as feasible kernels can be constructed.

References

[1] J. Yuan, C. Pixley, A. Aziz, and K. Albin. A Framework for constrained functional verification. *ICCAD*, 2003.

[2] Aharon Aharon, et al., Test Program Generation for Functional Verification of PowerPC Processors in IBM. Proc. *DAC*, 1995.

[3] Aharon Aharon, et al., Verification of the IBM RISC System16000 by a dynamic biased pseudo-random test program generator. *IBM Systems J.*, Vol 30, No 4, pp. 527 - 538, 1991.

[4] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets, A study in coverage-driven test generation. Proc. *DAC*, pp. 970 - 975, 1999.

[5] G. Nativ, S. Mittermaier, S. Ur, and A. Ziv, Cost evaluation of coverage directed test generation for the IBM mainframe. Proc. *ITC*, pp. 793 - 802, 2001.

[6] P. Mishra and N. D. Dutt, Functional coverage driven test generation for validation of pipelined processors. Proc. *DATE*, pp. 678 - 683, 2005.

[7] S. Fine and A. Ziv, Coverage directed test generation for functional verification using Bayesian networks. Proc. *DAC*, 2003.

[8] Onur Guzey and Li-C. Wang, Coverage-directed test generation through automatic constraint extraction. Proc. *IEEE HLDVT*, 2007.

[9] Mango C.-T. Chao, et al. Pattern selection for testing of deep sub-micron timing defects. *DATE*, pp. 1060 - 1065, 2004.

[10] Benjamin Lee, et al. Reducing Pattern Delay Variations for Screening Frequency-Dependent Defects. IEEE *VTS*, 2005.

[11] Onur Guzey, Li-C Wang, Jeremy Levitt and Harry Foster, Functional test selection based on unsupervised support vector analysis *DAC*, pp. 262 - 267, 2008.

[12] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*, Cambridge Univ. Press, 2004.

[13] Bernhard Schlkopf, and Alexander J. Smola. Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, 2001.

[14] Pang-Ning Tan, Michael Steinbach, Vipin Kumar *Introduction to Data Mining*, Addison Wesley, 2005

[15] PLASMA at http://www.opencores.com/project,plasma

[16] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini and C. Watkins, Text classification using string kernels, *J. Mach. Learn. Res. 2*, pp. 419 - 444, 2002.

[17] K. Riesen, M. Neuhaus, and H. Bunke, Bipartite graph matching for computing the edit distance of graphs, *Graph-Based Representations in Pattern Recognition*, 2007.

[18] R. Wilson and E. Hancock, Structural matching by discrete relaxation, *IEEE Tran. on Pattern Analysis and Machine Intelligence*, Vol 19, pp. 634 - 648, 1997.