Accepted for publication in the 13th Reconfigurable Architectures Workshop (RAW 2006) at IPDPS 2006, April, 2006, Rhodes Island, Greece.

Dynamic Configuration Steering for a Reconfigurable Superscalar Processor

Nick A. Mould[1], Brian F. Veale[2], Monte P. Tull[1], and John K. Antonio[2]

[1]University of Oklahoma
School of Electrical and Computer Engineering
Norman, OK 73019-1023 USA
{nick_mould, tull}@ou.edu

[2]University of Oklahoma
School of Computer Science
Norman, OK 73019-6151 USA
{veale, antonio}@ou.edu

Abstract

A new dynamic vector approach for the selection and management of the configuration of a reconfigurable superscalar processor is proposed. This new method improves on previous work that used steering vectors to guide the selection of functional units to be loaded into the processor. Dependencies among instructions in the instruction buffer are analyzed to enable a new scoring method. The dynamic vector technique is shown to reduce the amount of reconfiguration required while preserving execution resources. Simulation results reveal that, given enough configurable space, the configuration of the processor approaches a stable state.

1. Introduction and Related Work

This paper builds on work presented in [1] where a configuration management controller for a reconfigurable superscalar processor is proposed. A main goal of the extensions proposed and studied in this paper is to optimize the usage of the reconfigurable resources available in the processor proposed in [1].

The architecture assumed in this paper is similar to that presented in [1] and originally proposed in [2]. This architecture is partially run-time reconfigurable at the level of reconfigurable functional unit (RFU) "slots". The architecture of [1] provides eight RFU "slots" and a number of fixed functional units (FFUs) that cannot be reconfigured. As a program is executed, the configuration manager loads execution units into the RFU slots, as needed. Each type of execution unit that can be loaded requires one or more RFU slots. Five types of execution units are included: (1) integer arithmetic/logic units (Int-ALUs), (2) integer multiply/divide units (Int-MDUs), (3) load-store units (LSUs), (4) floating-point arithmetic/logic units (FP-ALUs), and (5) floating-point multiply/divide units (FP-MDUs) [1]. Table 1 lists the number of slots required to support each type of execution unit.

The main contribution of [1] is a configuration management system that determines when the processor should be reconfigured, by RFU type and quantity. The method proposed in [1] uses a "Configuration Selection Unit" that chooses between one of four possible pre-defined configurations (referred to as "steering vectors"). The RFUs defined by the chosen steering vector are configured into idle RFU slots provided in the processor [1].

The Configuration Selection Unit determines which steering vector to use based on a "configuration error metric" (CEM) that compares the needs of the instructions in the instruction buffer with that of three pre-defined steering vectors (shown in Table 1 as Configurations 1 – 3) as well as the current configuration. The steering vector, or the current configuration, that is closest (i.e., that has the smallest CEM value) to the needs of the instructions in the instruction buffer is chosen and loaded into available RFU slots. If a configured RFU is currently busy executing an instruction, it is not reconfigured. Thus, at any point in time, the current configuration of the RFUs contains a mixture of the execution unit combinations specified by the three steering vectors [1].

The number of execution units of each type that are specified by the steering vectors is provided in Table 1. Note that in [1], the steering vectors not only specify the types and quantities of execution units that can be loaded into the RFU slots, but also specify where they can be loaded into the RFU slots. Additionally, to prevent stalling, in the case where a required execution unit is never loaded, a set of five fixed functional units (FFUs) consisting of one of each type of execution units is provided in the architecture of [1].

The CEM calculation of [1] considers all instructions in the instruction buffer regardless of instruction status (unassigned and ready for execution, assigned and executing, waiting on dependencies, or independent). Thus, priority is not given to those instructions whose

Table 1. Number of each type of functional unit provided in [1] and the number of RFU slots required for each type, derived from [1].

	Int-ALU	Int-MDU	LSU	FP-ALU	FP-MDU
# of RFU Slots Required	2	2	1	3	3
RFUs – Configuration 1	1	1	4	0	0
RFUs – Configuration 2	0	0	2	1	1
RFUs – Configuration 3	2	2	0	0	0
RFUs – Configuration 0 (Current)	0 - 2	0 - 3	0 - 4	0 - 1	0 - 1
FFUs	1	1	1	1	1

dependencies have been met and have not been scheduled. This can result in the processor disregarding instructions that may be on a critical path of flow through the program, thereby degrading the processor performance. By contrast, the approach introduced in this paper proposes a new procedure based on dynamic vector construction (DVC) that prioritizes instructions such that satisfaction of their resource requirements is guaranteed, thereby eliminating the need for fixed units. Also, the proposed approach considers the effect of dependencies between instructions to make more effective use of reconfigurable resources. For example, a dependent chain of three instructions that all require the same type of RFU are recognized as needing only one RFU of that type, and not three. The scoring approach in [1] does not consider dependencies among instructions and thus would deduce that three instructions of the same type require three RFUs, even though the linear structure of the dependency chain may not admit the assumed parallelism.

The remainder of the paper is organized as follows: Section 2 introduces formal developments related to reconfiguration complexity and the steering vector based approach of [1]; Section 3 proposes and evaluates extensions to the steering vector approach of [1]; Section 4 reviews previous work related to the new DVC procedure; Section 5 details the DVC procedure; Section 6 presents an experimental study of the new DVC procedure.

2. Configuration Space Complexity

For the static steering vector method of [1] it is important to assure that all possible combinations (or all desired combinations) of RFUs are achievable through proper selection of steering vectors. The analysis provided in this section provides results and conditions related to the satisfaction of this objective.

For the purposes of our analysis, we assume a finite reconfigurable space of integer size, and we further assume that the size of the RFU's is of integer measurement and, without loss of generality, that the size of the smallest RFU is unity. For a given collection of steering vectors, there exist a finite number of possible permutations of the RFUs that can ultimately populate the configurable space. Recall that the approach of [1] yields a current configuration that is generally a combination of the steering vector components. This is because a selected steering vector is generally only partially loaded, i.e., only those vector elements (RFUs) associated with available slots are loaded.

Some of the resulting permutations are equivalent in the sense that they contain the same number of RFUs of each type. We will refer to each set of equivalent permutations as a unique combination. The number of unique combinations can be calculated directly from the size of the reconfigurable space and the size of each RFU considered. Let N denote the (integer) size of the reconfigurable space and let E be an n-tuple vector where each element e_1, e_2, e_3,…, e_n designates the integer size of n possible RFU types. Finally, let the vector $K = \langle k_1, k_2, k_3, ..., k_n \rangle$ represent the multiplicity of each RFU type present in a given combination. With these definitions, the number of unique combinations is equal to the number of nonnegative integer solutions to Equation (1), which is expressed in component form in Equation (2). As stated earlier, we assume a minimal RFU size of unity, which implies that all combinations are complete in the sense that "wasted space," does not exist.

$$E \bullet K = N \tag{1}$$

$$k_1 e_1 + k_2 e_2 + k_3 e_3 + \cdots + k_n e_n = N \tag{2}$$

The number of nonnegative integer solutions to Equation (2) may be found either iteratively or with the clever use of a power series representation, as illustrated by Example 1.

Example 1. Calculation of the number of unique combinations with E = <1,2,2,3,3> and N = 8.

Recall that k_1, k_2, k_3, k_4, and k_5 are the multiplicity of each RFU type in a given combination. Then from Equation (2):

$$k_1 + 2k_2 + 2k_3 + 3k_4 + 3k_5 = 8 \tag{3}$$

The number of unique combinations is exactly equal to the number of nonnegative integer solutions to Equation (3). Although an iterative method for determining the solutions is possible, a more convenient way to count the number of solutions is to use a power series representation. The identity relation in Equation (4) can be used to derive Equation (5) [3].

$$\left(\sum_{i=0}^{\infty} x^i\right)^2 = \sum_{n=0}^{\infty}(n+1)x^i \qquad \text{If } |x|<1 \tag{4}$$

$$\sum_{i=0}^{\infty}(i+1)x^{2i} = (1+2x^2+3x^4+4x^8+\cdots) \tag{5}$$

Consider the exponent, $2i$, on the left hand side of Equation (5) to represent the amount of available reconfigurable space. Further, for the sake of discussion, assume we wish to fill this space with two different elements, each of size two, then the coefficient, i+1, represents the number of unique ways that the space can be constructed. A power series representation of our specific example is shown in Equation (6). S_i is the number of ways in which we can fill a space of size i using one element of size one, two elements of size two, and two elements of size three. The goal is to find S_8, the coefficient proceeding x^8 on the left hand side of Equation (7).

$$\sum_{i=0}^{\infty} S_i x^i = \left(\sum_{k_1=0}^{\infty} x^{k_1}\right)\left(\sum_{k_2=0}^{\infty} x^{2k_2}\right)\left(\sum_{k_3=0}^{\infty} x^{2k_3}\right)\bullet \left(\sum_{k_4=0}^{\infty} x^{3k_4}\right)\left(\sum_{k_5=0}^{\infty} x^{3k_5}\right) \tag{6}$$

Use of Equation (4) reduces Equation (6) into a compact form given by Equation (7), where $a=k_1$, $b=k_2+k_3$, and $c=k_4+k_5$:

$$\sum_{i=0}^{\infty} S_i x^i = \left(\sum_{a=0}^{\infty} x^a\right)\left(\sum_{b=0}^{\infty}(b+1)x^{2b}\right)\bullet \left(\sum_{c=0}^{\infty}(c+1)x^{3c}\right) \tag{7}$$

Because there are many ways to obtain an exponent of eight using the exponents on the right hand side of Equation (7), we must iterate through them to determine the coefficients whose sum is S_8.

For $c=2$, an exponent of size six is produced:

$$\left(\sum_{a=0}^{\infty} x^a\right)\left(\sum_{b=0}^{\infty}(b+1)x^{2b}\right)\left(3x^6\right) \tag{8}$$

To obtain an exponent of size eight, we can either set b=1 and a=0, or b=0 and a=2. The results, respectively:

$$(x^0)(2x^2)(3x^6) = 6x^8 \tag{9}$$

$$(x^2)(x^0)(3x^6) = 3x^8 \tag{10}$$

Now, with c=1, an exponent of size three is produced:

$$\left(\sum_{a=0}^{\infty} x^a\right)\left(\sum_{b=0}^{\infty}(b+1)x^{2b}\right)(2x^3) \tag{11}$$

To obtain an exponent of size eight, we can either set b=2 and a=1, or b=1 and a=3, or b=0 and a=5. The results again, respectively:

$$(x)(3x^4)(2x^3) = 6x^8 \tag{12}$$

$$(x^3)(2x^2)(2x^3) = 4x^8 \tag{13}$$

$$(x^5)(x^0)(2x^3) = 2x^8 \tag{14}$$

With c = 0 an exponent of size eight now becomes a full iteration through the variable b. The values are shown here along with the obtained coefficients from the right hand side of Equation (7).

$$\left(\sum_{a=0}^{\infty} x^a\right)\left(\sum_{b=0}^{\infty}(b+1)x^{2b}\right)(x^0) \tag{15}$$

For b = 4, a = 0:

$$(x^0)(5x^8)(x^0) = 5x^8 \tag{16}$$

For b = 3, a = 2:

$$(x^2)(4x^6)(x^0) = 4x^8 \tag{17}$$

For b = 2, a = 4:

$$(x^4)(3x^4)(x^0) = 3x^8 \tag{18}$$

For b = 1, a = 6:

$$(x^6)(2x^2)(x^0) = 2x^8 \tag{19}$$

For b = 0, a = 8:

$$(x^8)(x^0)(x^0) = x^8 \tag{20}$$

S_8 is the sum of all the coefficients obtained in the construction of exponents of size eight on the right hand side of Equation (7) as shown in Equations (9, 10, 12-14, and 16-20).

$$S_8 = 6+3+6+4+2+5+4+3+2+1 = 36$$

In this specific example configuration space, thirty-six possible unique execution unit combinations are possible. If steering vectors are selected properly, then thirty-six unique configurations would be possible during run-time.

In order to develop a set of steering vectors that can reach all of the unique combinations of the RFU types, first observe that within the entire set of unique combinations there exists a subset of combinations in which only one type of RFU appears in each. The size of this subset is equal to the number of RFU types. This set of vectors forms a basis for reaching every unique combination. Other valid basis sets can be derived from this subset by interchanging RFUs among the steering vectors, provided that the slots inhabited by a given RFU type remain disjoint from slots inhabited by other RFU's of the same type.

3. Steering Vector Score Analysis

The steering vector score as proposed in [1] can be improved by (1) simply scoring only those instructions that are ready for assignment. However, this may result in an overreaction to the instructions' actual needs causing the machine to thrash about in a constant state of reconfiguration. There are other scoring methods possible, such as: (2) scoring only instructions that are unassigned, (3) scoring only instructions that are ready (dependencies met), and (4) scoring only instructions that are dependent.

To study the steering vector scoring alternatives, the Susan benchmark from the Automotive and Industrial Control category of the MiBench set of embedded benchmarks was chosen. The Susan benchmark is an application that is used to detect corners and edges in images [4]. The benchmark is traced on a PowerPC based machine and the resulting trace is simulated using a software-based simulator of the scoring technique. This simulator determines when the RFUs of the processor should be reconfigured and when to issue instructions to the FFUs and RFUs. Various statistics are gathered by the simulator such as the total estimated clock cycles to complete the benchmark and the average usage of the FFUs and RFUs.

Additionally, RFUs required in the first level of the directed acyclic graph (DAG) may also be required for levels further into the DAG; however, the score in [1] assumes the resource requirement should be the sum of all the resources needed by each level in the DAG. This is inaccurate since the resource needs are actually the maximum number of resources required by each level and type, calculated as the maximum of each resource type required by each level. An example of this calculation is shown in Figure 1.

The simulation results shown in Table 2 suggest that the fixed RFU's perform the bulk of the processing. It is also apparent that the configurable space is sparsely used, and therefore unable to impact the total execution time in any discernable manner with regard to the specific scoring method being employed.

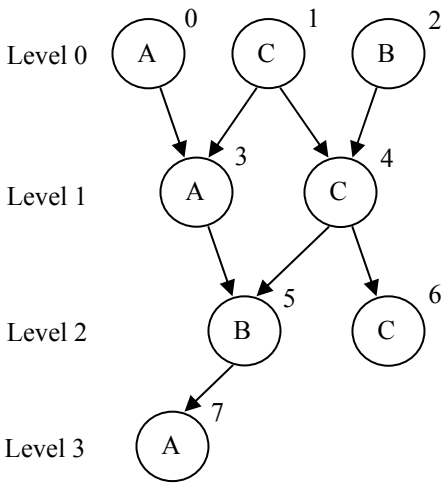
Table 2 also suggests that some RFUs are configured but never used. To overcome the deficiencies of these scoring methods, a new dynamic vector construction (DVC) procedure is developed in Section 5 that performs resource allocation according to a DAG level analysis.

4. Related Work in Dynamic Vector Construction

The new DVC procedure presented in Section 5 provides an improved scoring method using a level analysis of the DAG, the results of which are used by a dynamic vector update (DVU) procedure that configures a resource vector. The DVC method concerns the mapping of a sub-DAG contained within an instruction buffer onto a set of dynamically changing resources. In the past, a number of solutions have been proposed for the mapping of a DAG onto a set of fixed resources, where the resources have traditionally been a static set of heterogeneous processors [5-8].

Table 2. Simulation results with steering vectors and FFUs, varying score method, varying instruction buffer size, a reconfiguration time of eight clock cycles, and a reconfiguration space of eight RFU Slots.

Score	Instruction Buffer Size	Total Clock Cycles	AVG Fixed Unit Usage per Cycle	AVG Reconfigurable Unit Usage per Cycle
Assigned and Unassigned Instructions	4	63335875	1.002042	0.283295
	8	50585521	1.405507	0.207679
	12	45410010	1.603518	0.204732
	16	45911824	1.619694	0.18068
Unassigned Instructions	4	63483253	1.105618	0.177432
	8	50349612	1.416548	0.198542
	12	45860338	1.613274	0.178723
	16	45964526	1.608275	0.188117
Instructions Ready for Execution	4	63191348	1.19228	0.100043
	8	50423021	1.506096	0.109673
	12	45860883	1.689452	0.091539
	16	46277048	1.650722	0.115905
Dependent Instructions Only	4	63868305	0.983803	0.288484
	8	50379426	1.29472	0.320752
	12	45847016	1.66918	0.109271
	16	46802729	1.620023	0.12835



RFU Need by Level:

	A	B	C
Level 0	1	1	1
Level 1	1	0	1
Level 2	0	1	1
Level 3	1	0	0

Calculated DAG Need:

	A	B	C
Without Level Analysis	3	2	3
With Level Analysis	1	1	1

Figure 1. Reconfigurable functional unit need calculation comparison.

Previously, DAG mapping is cast as a two part problem where the solution, usually either a priority based list [5] or a dynamically calculated heuristic [5, 6], encompasses both the mapping and the scheduling of instructions from the DAG onto a set of static resources [5-7]. For the case of static resources, mapping and scheduling are dependent and cannot be broken down into independent sub-problems [5]. However, when a set of instructions are to be scheduled and mapped onto a set of dynamically changing resources, as is the problem of concern in this paper, scheduling and mapping solutions can be considered independent.

The work in [5] claims that mapping and scheduling are dependent since all instructions in a specific DAG level are forced to depend on all instructions in the previous DAG level. The advantage of using dynamically reconfigurable resources is that resource need of an instruction is a function of its position in the DAG. A level analysis procedure can be used to create a near optimal resource map that can be dynamically updated to correspond to the changing needs of incoming instructions. This resource map creation allows the DVC procedure to support instruction level parallelism (ILP) as well as resource allocation for the critical path.

The work in [6] focuses on best matching of instructions to resources with the use of a generalized dynamic level (GDL) calculation; however, GDL does not prioritize based on dependencies and therefore no priority is given to ILP opportunities.

In [7], ILP is exploited using either counting or bit vector algorithms that dynamically analyze the instructions in the DAG and place ready instructions into a processing queue. Since reconfiguration is not considered, there is no attempt to analyze the DAG by level and detect future ILP.

5. DVC Procedure

This section proposes a new technique for the determination of resources required to support instructions that are in the instruction buffer. This technique analyzes the sub-DAG of the program being executed that consists of the instructions present in the instruction buffer. Section 5.1 discusses how each level of this sub-DAG is analyzed to identify the resources necessary to exploit the inherent ILP of each level. Section 5.2 presents the dynamic vector update procedure that uses the results of this analysis to prioritize RFUs, by level, in order to determine which RFUs should be loaded into or removed from the current configuration.

5.1. Level Analysis Procedure

The dependencies between instructions in the instruction buffer can be represented by the dependency matrix D. Note that D is of square dimension and is of size $n \times n$, where n is equal to the number of instructions. Any element of D, d_{ij}, having a logic value of one indicates that instruction i is dependent upon the completion of instruction j; otherwise, d_{ij} has a logic value of zero. A procedure is presented that transforms the instruction dependency matrix D into a level readiness matrix T, where any element, t_{ij}, having a logic value of one indicates that instruction j is a member of level i.

For convenience in transforming matrix D into matrix T, an intermediate matrix M is used, where each element in M, m_{ij}, having a logic value of one indicates that instruction j is a member of level less than or equal to i.

Observing that any instructions that depend solely on those that are members of level zero must be members of level one. Additionally, all instructions in level one must be dependent upon at least one instruction that is a member of level zero. Thus, it is apparent that if the projection of row i of D onto row j of M is equal to row i of D, instruction i must depend on a level less than or

equal to j. Following this logic, matrix M is created, as specified in Equation (20).

$$m_{i,j} = \begin{cases} \overline{\sum_{k=0}^{n-1} d_{i,k}} \ \text{if} \ i = 0 \\ \overline{\sum_{k=0}^{n-1} \left\{ (m_{i-1,k} \bullet d_{j,k}) \oplus d_{j,k} \right\}} \ \text{if} \ 0 < i < n \end{cases} \quad (20)$$

Note that M defined by Equation (20) is complete. The final step in the transformation from $D \rightarrow T$ is shown in Equation (21). This equation applies an XOR operation across the columns of M to separate the rows of M.

$$t_{i,j} = \begin{cases} m_{i,j} \ \text{if} \ i = 0 \\ m_{i,j} \oplus m_{i-1,j} \ \text{if} \ 0 < i < n \end{cases} \quad (21)$$

The intermediate matrix, M, is not necessary for implementation, and T can be computed directly from D using a combinational circuit.

5.2. Dynamic Vector Update Procedure

Given the results obtained through level analysis of the DAG and information on the specific resource requirements of any given instruction, the exact resources necessary for exploiting all of the ILP for any given sub-DAG can be determined. A priority-based scheduling solution exists provided that the allocated resource space is at least as large as the largest RFU. We assume that RFU slots can be reconfigured as necessary if contiguous available space exists that is greater than or equal to the size of the RFU being configured. The goals of the dynamic vector update (DVU) procedure are to (1) avoid loading unnecessary resources, (2) avoid discarding valuable resources, and (3) guarantee efficient execution of instructions along the critical path when possible.

Let R be a k element vector where each element is of size $\lceil \log_2 n \rceil$, and each element, r_j, represents the number of resources of type j necessary for satisfying instructions at the level currently being analyzed. Using R, the DVU procedure is shown in Figure 2.

Note that only unassigned instructions are considered to have RFU needs, and the DVU procedure only analyzes unassigned instructions.

To avoid loading of unnecessary resources, examination of resource needs by level guarantees that the possibility of reusing resources between levels is completely exploited. For example, if level zero requires two type A resources, and level three requires one type A resource, and no other levels require any type A resources, then the actual need for all levels is two type A resources. In contrast to the steering vector approach that would assume that three type A resources are necessary,

```
Initialize Configured Resources Priority to level n

Loop over levels i ∈ [0, n − 1]
  R = Count of resources types required by level i
  While the resource configuration space contains
    resources required by R
      Set resource priority to level i
      Decrement requirement in R because the resource
        already exists
  Loop over R  j ∈ [0, k − 1]
    If R(j) > 0
      Load Resource Type (i, j)
  End Loop
End Loop

Load Resource Type
  Loop over resource configuration space
    If Contiguous space exists
      If unused space exists
          Load Type j at current location
      Else if space is unused and designated for level
        greater than or equal to i
          Load Type j at current location
      Else Fail
  End loop
End Load Resource Type
```

Figure 2. Dynamic vector update procedure.

the level dependency analysis procedure makes efficient use of resources, as this analysis can detect that multiple levels can utilize the same resources over time.

To avoid discarding valuable resources, if there are unused RFU slots, the dynamic loading strategy will use those slots. Over time it becomes necessary to discard unused resources and load others. For example, if level one requires RFU type B and the RFU is not currently loaded into any slot, the procedure would allow a resource of type B to be loaded in any space that will not be used for levels zero or one. To guarantee that valuable resources are not discarded, any level being analyzed for resource loading can only discard resources that are not designated for use by a previous level, including the current level. This policy also has the effect of making the process of resource discarding priority-based.

The exploitation of ILP at level zero is limited by the size of the configuration space and the ability to locate contiguous available space for loading the required RFUs. Therefore, the instruction buffer and the configuration space should be carefully designed to ensure that the DVU procedure is able to utilize ILP. Example 2 shows the DVC level analysis procedure for the DAG given in Figure 1.

Example 2. Calculating Matrix T from Matrix D begins with a complete dependency matrix D, obtained, in this example, by analyzing Figure 1.

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

The first row of matrix M is calculated using Equation (20), where i=0. Elements $m_{0,0}$ and $m_{0,5}$ are shown in Equations (22 and 23), respectively. Equation (24) shows row zero of M.

$$m_{0,0} = \overline{\sum_{k=0}^{7} d_{0,k}} = 1 \tag{22}$$

$$m_{0,5} = \overline{\sum_{k=0}^{7} d_{5,k}} = 0 \tag{23}$$

$$M_{row(0)} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{24}$$

The result obtained in Equation (22) shows that instruction zero is a member of level zero. Equation (23) shows that instruction number 5 is not a member of level zero. All other rows of M are calculated using the second part of Equation (20). Equation (25 - 27) show the calculation of $m_{1,4}$.

$$m_{1,4} = \overline{\sum_{k=0}^{7} \left\{ (m_{0,k} \bullet d_{4,k}) \oplus d_{4,k} \right\}} \tag{25}$$

$$m_{1,4} = \{[1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0] \bullet \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\} \oplus \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{26}$$

$$m_{1,4} = 1 \tag{27}$$

Calculating each element, shown in Equations (25-27) results in M, shown in Equation (28):

$$M = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{28}$$

Matrix T is determined using Equation (21). Calculation of $t_{1,3}$ and $t_{1,6}$ are shown in Equations (29) and (30), respectively.

$$t_{1,3} = m_{1,3} \oplus m_{0,3} = 1 \tag{29}$$

$$t_{1,6} = m_{1,6} \oplus m_{0,6} = 0 \tag{30}$$

The result shown in Equation (29) indicates that instruction number 3 is a member of level one. The result shown in Equation (30) indicates that instruction number 6 is not a member of level one.

Examination of Figure 1 confirms the results obtained in Equations (29) an (30), as well as the final matrix T shown below in Equation (31).

$$T = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{31}$$

6. Experimental Results

This section presents an experimental study of the DVC procedure proposed in Section 5. This study evaluates the procedure on the basis of execution time and RFU slot usage using a software simulator that simulates the Configuration Manager of [1] modified to use the DVC procedure instead of a steering vector based approach. This simulator permits the size of the configuration space, the time required to reconfigure an RFU slot, and the instruction buffer size to be specified by the user.

The study presented in this section examines the performance of the DVC procedure on the same Susan benchmark [4], which was used in the evaluation of the steering vector scoring technique in Section 3. The simulator reads the Susan benchmark trace, reconfigures the processor and assigns instructions according to the DVC procedure. As a final output, for a given buffer size, the simulator identifies the optimal number of execution units by type and quantity.

Figure 3 shows that as the configuration space size is increased, the reconfiguration time becomes less of a factor in the overall performance of the machine. When given enough configuration space, the DVC procedure converges to a near-optimal configuration and remains stable, thereby eliminating the need for further reconfiguration and reducing the time penalty incurred due to reconfiguration.

Figure 4 shows that as the DVC converges to the optimal configuration, units (RFU slots) are no longer configured and not used, i.e., wasted. Note that units are wasted when there is not enough configurable space to load units in advance. This is due to the fact that if there is

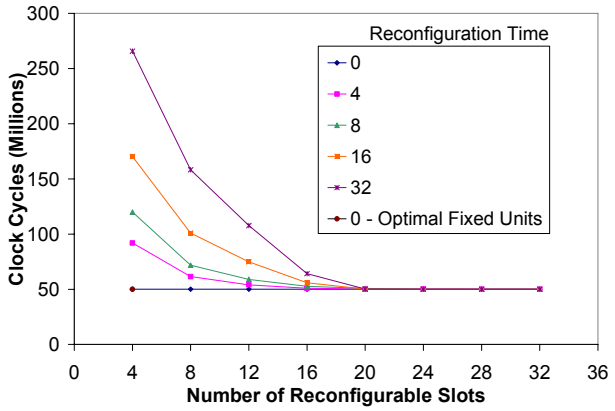


Figure 3. Total execution time in clock cycles versus configuration space size measured in slots, with an instruction buffer of size of eight.

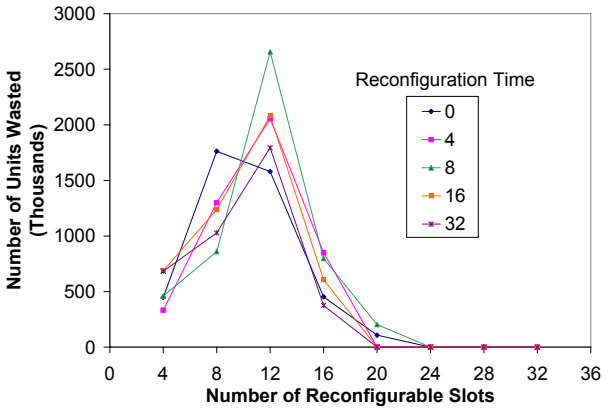


Figure 4. Number of units wasted versus configuration space size, measured in slots, with an instruction buffer of size of eight.

not enough space, then some unit must be preempted when another unit of higher priority is chosen to be loaded.

The results obtained through simulation suggest that when given enough configuration space, the DVC procedure causes the combination of RFU's in the configuration space to converge to a stable and near-optimal configuration. When the configurable space is in an optimal configuration, the number of units wasted approaches zero. Also, if the DVC procedure converges quickly, then the overall reconfiguration time is minimized.

7. Conclusions

A new approach for the selection and management of the configurable space of a reconfigurable superscalar processor is proposed. This approach is based on a sub-DAG level analysis of the instructions in the instruction buffer of the processor. This analysis determines the priority of functional units to be loaded into the configurable space according to their dependency level in the sub-DAG. The method is shown to provide efficient utilization of configurable resources resulting in a minimal amount of reconfiguration.

8. References

[1] Veale, B.F., Antonio, J.K., and Tull, M.P., "Configuration Steering for a Reconfigurable Superscalar Processor," *12th Reconfigurable Architectures Workshop (RAW 2005), Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Apr. 2005.

[2] Niyonkuru, A. and Zeidler, H.C., "Designing a Runtime Reconfigurable Processor for General Purpose Applications," *Reconfigurable Architectures Workshop (RAW 2004), Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, pp. 143–149, Apr. 2004.

[3] Stewart, J., *Calculus*, 5th Edition, Thomson Brooks-Cole, Pacific Grove, CA, 2003.

[4] Guthaus, M.R., Ringenberg, D.E., Austin, T.M., et al, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Proceedings of the 4th Annual IEEE Workshop on Workload Characterization*, pp. 3-14, Dec. 2001.

[5] Ahmad, I., Dhodhi, M.K., and Ul-Mustafa, R., "DPS: Dynamic Priority Scheduling Heuristic for Heterogeneous Computing Systems," *IEE Proceedings of Computers and Digital Techniques*, Vol. 145, Iss. 6, pp. 411–418, Nov. 1998.

[6] Sih, G.C. and Lee, E.A., "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, Iss. 2, pp. 175–187, Feb. 1993.

[7] Beckmann, C.J., and Polychronopoulos, C.D., "Microarchitecture Support For Dynamic Scheduling Of Acyclic Task Graphs," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 140–148, Dec. 1992.

[8] Kwok, Y.-K., and Ahmad, I., "Benchmarking the task graph scheduling algorithms," *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998)*, pp. 531–537, Apr. 1998.