A Comparative Analysis of the Hardware Requirements for the Lagrange-Euler and Newton-Euler Dynamics Formulations[1]

Pradeep K. Khosla[2] and Sandra Ramos[3]

Abstract

The improved performance of model-based control schemes for manipulators has demonstrated the need for including a dynamical model in the control law. This requires that the inverse dynamics be computed at real-time sampling rates of about 500 Hz. There are two major formulations which can be used to model the robot arm dynamics: the Newton-Euler (N-E) formulation and the Lagrange-Euler (L-E) formulation. Due to their large computational requirements a challenging task has been to devise alternate methods of reducing their computational cycle. Further, the proposed methods should result in a feasible implementation that minimizes the hardware requirements for the inverse dynamics computation. In our previous work, we have proposed a parallel computational scheme that is based on the *mathematical decomposition* of the equations into their primitive matrix/vector arithmetic operations. We have shown that the mathematical decomposition scheme provides an efficient mechanism to reduce the computational cycle of both the L-E and N-E formulations. In this paper, we analyze the N-E and L-E equations from a hardware perspective and compare the results for each. Our analysis shows that N-E is indeed more efficient than L-E from the computational as well as the hardware point of view.

1. INTRODUCTION

Model-based control of manipulators requires computing the dynamics equations, which are computation intensive, in real-time [4]. Researchers have concentrated their efforts on the minimization of the dynamics computational cycle of the Newton-Euler(N-E) formulation due to its computational advantages over the Lagrange-Euler(L-E) formulation. In addition, dynamics formulations other than N-E and L-E have been proposed [7, 6]. Some of the optimization methods proposed for the real-time computation of the NE formulation include: customization of the equations for a given manipulator [5, 3, 2], the use of table-lookup methods [13, 14], the design of special-purpose processors [8], and the design of special-purpose parallel architectures [11, 10, 12]. In particular, the recently enhanced performance of floating point processors and the inherent structure of the recursive formulations makes implementing the dynamics equations on parallel architectures an attractive alternative for decreasing the computation time.

Researchers have proposed various multiprocessor architectures to compute the recursive NE equations. The design approaches of the proposed architectures differ on how to split the computations for parallel execution[4]. The decomposition schemes proposed for inverse dynamics are based on the allocation of functional subsets of the computations to parallel processors. These functional subsets can take many different forms. For example, one decomposition scheme explores the allocation of processors to functional subsets of operations pertaining to a manipulator joint, leading to a configuration-dependent architecture [10]. Another approach decomposes the computational space into modular subsets each of which are dedicated to the computation of a dynamic variable [11]. Dedicating processors to such high-level functional tasks leads to low processor utilization since the execution of these tasks are not evenly distributed throughout the computational cycle. A more efficient architecture would be one based on the decomposition of the computations into functional subsets of a more elementary nature, such as the computation of fundamental matrix/vector operations. This is the underlying idea of the decomposition scheme proposed in [17]. This scheduling approach is called the **Mathematical Decomposition** scheme and has been shown to be an efficient technique for exploiting parallelism in the equations of motion of a serial link manipulator [17].

The main thrust of this paper is to present a comparative analysis of the computational hardware requirements for the L-E and N-E formulations. To achieve our goal, we have divided our analysis into three stages. First, we apply the mathematical decomposition technique to the L-E and N-E equations to obtain *efficient scheduling schemes*[5] for the parallel computations required by each formulation. Second, we analyze the scheduling schemes obtained for both formulations in the first stage, and we obtain worst-case hardware requirements for each case. Third, we compare the hardware requirements for the L-E and N-E formulations, as well as their computational cycles. Our results show that the N-E formulation is more efficient than the N-E formulation both, from the hardware and the software perspective.

2. DYNAMICS FORMULATIONS

In this section, we present both the L-E and the N-E equations for an *n* degress-of-freedom manipulator. Because of lack of space, we have included only the equations without any detailed explanation. For further details the reader is referred to the original articles for the L-E equations [1] and for the N-E equations [9].

The 3x3 Lagrange-Euler equations, depicted in Table 2-1, consist of two sets of recursions: forward recursions and backward recursions. For the purpose of our analysis, we will henceforth refer to the computation of each equation (for all the links) as a **Computational Chain** (or CC). For example, the computation of W_i (for $i = 1,...,n$) constitutes a computational chain.

The sequential implementation of the 3x3 L-E equations for a typical 6 degrees-of-freedom manipulator (i.e., n=6) involves 2195 multiplications and 1719 additions, or a total of 3914 floating point operations [1].

[1]This research has been funded by the Cooperative Research Fellowship Program sponsored by AT&T Bell Laboratories and by the Electrical and Computer Engineering Department at Carnegie Mellon University

[2]Assistant Professor, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213

[3]Graduate student, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213

[4]For expository purposes, we have called the process of decomposing the equations a **decomposition scheme**.

[5]We define an efficient scheduling scheme as a scheduler of parallel computations which leads to a high processor utilization.

CH2555-1/88/0000/0291$01.00 © 1988 IEEE

$$W_i = W_{i-1}A_i \quad (1)$$

$$\dot{W}_i = \dot{W}_{i-1}A_{i-1} + W_{i-1}\dot{A}_{i-1} \quad (2)$$

$$\ddot{W}_i = \ddot{W}_{i-1}A_i + 2\dot{W}_{i-1}\frac{\partial(A_i)}{\partial(q_i)}\dot{q}_i + W_{i-1}\frac{\partial(^2A_i)}{\partial(q_i)^2}\dot{q}_i^2 + W_{i-1}\frac{\partial(A_i)}{\partial(q_i)}\ddot{q}_i \quad (3)$$

$$\ddot{p}_i = \ddot{p}_{i-1}^T - \ddot{W}_i p_i' \quad (4)$$

$$D_i = A_{i+1}D_{i+1} + p_{i+1}e_{i+1} + n_i\ddot{p}_i^T + J_i\ddot{W}_i^T \quad (5)$$

$$e_i = e_{i+1} + m_i\ddot{p}_i^T + n_i^T\ddot{W}_i^T \quad (6)$$

$$c_i = m_i r_i + A_{i+1}c_{i+1} \quad (7)$$

$$f_i = Tr\{\frac{\partial(W_i)}{\partial(q_i)}D_i\} - g^T\frac{\partial(W_i)}{\partial(q_i)}c_i \quad (8)$$

$\dot{W}_0 = \ddot{W}_0 = D_{n+1} = c_{n+1} = \mathbf{0}$ Initial Conditions

Table 2-1: Recursive 3x3 Lagrange-Euler Dynamic Equations

The Newton-Euler formulation [9], depicted in Table 2-2, consists of two sets of recursions: the forward recursions that transform the kinematic parameters from the base to the end-effector and the backward recursions that transform the dynamics variables and compute the joint torques.

$$\omega_{i+1} = \begin{cases} \mathbf{A}_{i+1}^T[\omega_i + \mathbf{z}_o\dot{\theta}_{i+1}] & \text{rotational} \\ \mathbf{A}_{i+1}^T\omega_i & \text{translational} \end{cases} \quad (9)$$

$$\dot{\omega}_{i+1} = \begin{cases} \mathbf{A}_{i+1}^T[\dot{\omega}_i + \mathbf{z}_o\ddot{\theta}_{i+1} + \omega_i\times(\mathbf{z}_o\dot{\theta}_{i+1})] & \text{rotational} \\ \mathbf{A}_{i+1}^T\dot{\omega}_i & \text{translational} \end{cases} \quad (10)$$

$$\dot{\mathbf{v}}_{i+1} = \begin{cases} \mathbf{A}_{i+1}^T\dot{\mathbf{v}}_i + \dot{\omega}_{i+1}\times\mathbf{p}_{i+1} + \omega_{i+1}\times(\omega_{i+1}\times\mathbf{p}_{i+1}) & \text{rotational} \\ \mathbf{A}_{i+1}^T[\dot{\mathbf{v}}_i + \mathbf{z}_o\ddot{d}_{i+1} + 2\omega_i\times(\mathbf{z}_o\dot{d}_{i+1})] + \dot{\omega}_{i+1}\times\mathbf{p}_{i+1} + \omega_{i+1}\times(\omega_{i+1}\times\mathbf{p}_{i+1}) & \text{translational} \end{cases} \quad (11)$$

$\omega_0 = \dot{\omega}_0 = \mathbf{v}_0 = \mathbf{0}$ Initial Conditions

$\dot{\mathbf{v}}_0 = [g_x g_y g_z]^T$ gravitational acceleration

$$\dot{\mathbf{v}}_i^* = \dot{\omega}_i\times\mathbf{s}_i + \omega_i\times(\omega_i\times\mathbf{s}_i) + \dot{\mathbf{v}}_i \quad (12)$$

$$\mathbf{F}_i = m_i\dot{\mathbf{v}}_i^* \quad (13)$$

$$\mathbf{N}_i = \mathbf{I}_i\dot{\omega}_i + \omega_i\times(\mathbf{I}_i\omega_i) \quad (14)$$

$$\mathbf{f}_i = \mathbf{A}_{i+1}\mathbf{f}_{i+1} + \mathbf{F}_i \quad (15)$$

$$\mathbf{n}_i = \mathbf{A}_{i+1}\mathbf{n}_{i+1} + \mathbf{p}_i\times\mathbf{f}_i + \mathbf{N}_i + \mathbf{s}_i\times\mathbf{F}_i \quad (16)$$

$$\tau_i = \begin{cases} \mathbf{n}_i^T(\mathbf{A}_i^T\mathbf{z}_o) & \text{rotational} \\ \mathbf{f}_i^T(\mathbf{A}_i^T\mathbf{z}_o) & \text{translational} \end{cases} \quad (17)$$

$\mathbf{f}_{N+1}$: external force at the end-effector.
$\mathbf{n}_{N+1}$: external moment at the end-effector.

Table 2-2: Newton-Euler Dynamic Equations

A sequential implementation of the N-E equations for a 6 degrees-of-freedom manipulator (DOF), all joints being rotational, requires 882 multiplications and 708 additions, or a total of 1590 floating point operations [16]. A uniprocessor implementation of the N-E formulation yields a computational cycle which is greatly limited by the total number of floating point operations and the operational speed of the microprocessor. Such limitations can be overcome by exploiting the parallelism inherent in the equations. The issue of parallelism is addressed in the next section.

3. PARALLELISM IN MANIPULATOR DYNAMICS

In partitioning the computations of a system of equations, there are many degrees of parallelism that can be exploited, each of which operates at a different level of abstraction. For example, **inter-chain parallelism** is the parallelism existing among computational chains, that is, among recursive equations. An example of inter-chain parallelism is the concurrent execution of operations in the recursive equations for $\ddot{W}_i$ (Equation(3)) and $\ddot{p}_i$ (Equation(4)). **Nodal parallelism**, on the other hand, involves the parallel execution of primitive vector arithmetic operations within the ith iteration of a given computational chain. An example of nodal parallelism is the parallel computation of the terms $A_{i+1}D_{i+1}$ and $J_i\ddot{W}_i^T$ in the calculation of D_i according to Equation (5). The lowest level of parallelism, called **operational parallelism**, is that which can be obtained at an operand level. For example, [a+b+c+d] can be computed as (a+b) and (c+d) in parallel, followed by [sum_{ab} + sum_{cd}], as opposed to [(((a+b) + c) + d)], which is the uniprocessor implementation.

The robot dynamic formulations presented in this paper consist of a group of computational chains which can be executed in parallel. These equations have a computational structure which permit the execution of parallel operations at all three levels: inter-chain, nodal, and operational. However, as the degree of parallelism increases, so do the hardware requirements to support the concurrent computations. If all three levels of parallelism are exploited, the hardware required to implement the dynamic equations becomes excessive. This is because a large number of basic arithmetic processing elements, such as floating point adders and multipliers must operate in parallel. In comparison, a much smaller number of processing elements are required when operational parallelism is neglected. For example, a single processor can multiply a 3x3 matrix with a 3x1 vector in 9 multiplications and 6 additions. However, if operational parallelism is exploited for every row of the matrix, the operation can be completed in the time it takes to execute 1 multiplication and 2 additions. However, it would require 9 parallel multipliers (one for each element of the matrix) and 3 adders. Due to the excessive hardware requirements of operational parallelism, only inter-chain and nodal parallelism are exploited in the hardware implementation presented in this paper.

4. ANALYZING HARDWARE REQUIREMENTS FOR MANIPULATOR DYNAMICS: APPROACH

The equations of motion of a manipulator involve a limited variety of vector arithmetic operations, such as vector cross product (**VC**), vector dot product (**VD**), matrix-vector multiplication (**MV**), etc. The mathematical decomposition technique treats every one of these operations as an atomic computation module[6]. For expository convenience, we classify these atomic computation modules into **primitive classes**, where a primitive class is a type of vector arithmetic operation, such as VC, VD, MV, etc. Each primitive class requires a given amount of floating point operations, such as additions and multiplications. In order to synchronize all concurrent processing, we define a **macrocycle** as the time it takes to compute the fastest one of these primitive classes (i.e., that one which has the least number of operations). In our case, a macrocycle is defined as the time required to add two 3x1 vectors, that is, a **VA** (vector addition) operation. Using a *macrocycle* as a basic time unit, allows us to normalize the time required to compute all the other primitive classes in terms of macrocycles. For example, if a 3x1 vector addition operation, requiring 3 floating-point operations (flops) is the fastest primitive class, then a macrocycle is defined as the time required to execute 3 flops. Consequently, a 3x1 vector cross product, requiring

[6]An atomic computation module is a subset of operations which are implemented sequentially.

9 flops, would take 3 macrocycles to be computed. The assignment of relative time weights to each primitive class in terms of number of macrocycles is one of the basic principles of the mathematical decomposition scheme.

The **Mathematical Decomposition** scheme consists of splitting the computation of an equation into a group of primitive matrix/vector operations and executing as many of these operations in parallel as possible. The objective is to solve the equation in the least number of stages. Applying this concept to a recursive equation and using the relative time weights assigned to each primitive class, we determine the number of macrocycles it takes to compute one iteration of the equation. Furthermore, this scheme allows us to calculate the total number of macrocycles it takes to complete an entire CC (i.e., all iterations). However, the measurements thus obtained for *a particular recursive equation* assume that the necessary data for each iteration will be available when needed. When considering *a group of recursive equations* such as the dynamic equations, this is not a valid assumption since there are data interdependencies amongst CC's (i.e., amongst the recursive equations). These data interdependencies may cause delays in the computations of a particular recursive equation if the primitive classes of such recursive equation must wait for the arrival of operands from other CC's which have not been computed yet. Thus, in using the mathematical decomposition scheme, we have to address the data interdependencies also. In order to achieve this goal, we graphically represent all parallel computational activity with reference to a common time scale by using an *activity chart*.

Activity charts provide a graphical representation of all the computational modules which are working in parallel at any instant of time as well as their data interdependencies. The are thus a valuable tool in assessing the time/space tradeoffs for various orderings of the computations. Since the computation time of any given atomic computation module (computational module) is that of the primitive class to which it belongs, these operations are all plotted with reference to a common time scale in terms of macrocycles. Hence, activity charts provide a good working environment from which the length of the computational cycle as well as the hardware requirements needed to support all parallel computations can be extracted. An activity chart for the 3x3 LE (of an 6 degrees-of-freedom manipulator) is depicted in Figure 4-1. This activity chart shows that the LE formulation comprises ten primitive matrix/vector operations. On the other hand, the activity chart of the N-E formulation, depicted in Figure 4-2, illustrates that it requires five different primitive matrix/vector operations. Our analysis shows that this is a crucial factor in determining the hardware utilization efficiency for these two formulations.

The process of determining computational hardware requirements consists of three steps. In the first step, we obtain **tabular charts** which indicate the number of computational modules of each primitive class that are needed at each instant of time throughout the computational cycle. In the second step, we interpret the data obtained in the first step and determine a preliminary processor count. The third and final step is a hardware minimization task in which we seek a reduction of peak computational demands. Each one of these steps is discussed in more detail in the sequel. Since the tabular charts for both formulations are fairly lengthy, they are not presented in this paper. These tabular charts are listed in [15]. However, for expository convenience we describe the process of creating a tabular chart with a simple example.

To obtain the data for the tabular chart corresponding to a given algorithm, we examine the number of parallel computations that have been allocated per macrocycle as depicted on its activity chart. For example, an entry in the tabular chart for the 40th macrocycle of the N-E formulation, is obtained by tracing a horizontal line across the activity chart at macrocycle 40, such as the one shown in Figure 4-2, and observing how many computational modules of each primitive class it intersects. The total number of active computational modules for macrocycles 40-44 is depicted in Table 4-1. The abbreviations used in Table 4-1 are explained in the activity chart of Figure 4-2. The

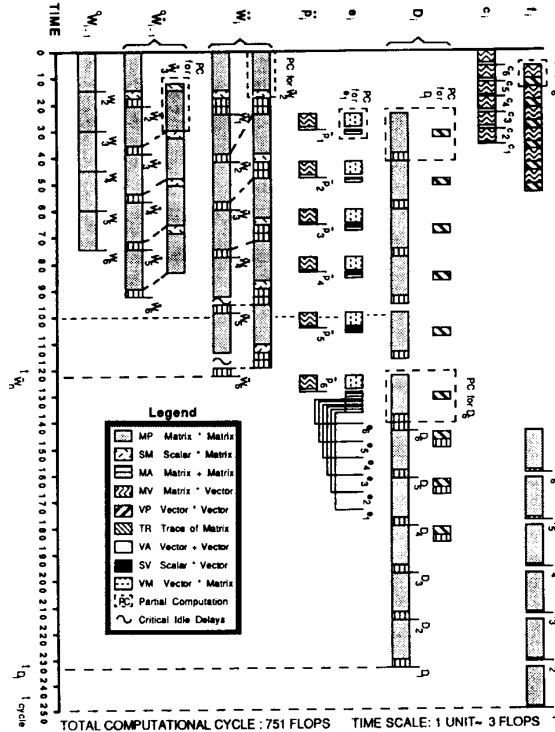

Figure 4-1: Activity Chart for 3x3 Recursive Langrange-Euler Case: N = 6

first step of the extraction process is completed once this procedure has been repeated over the entire computational cycle.

After data has been extracted from the activity chart, the information contained in the tabular chart must be processed in order to obtain the actual number of processing elements needed for an architectural implementation. The interpretation of such information is dependent on the type of target architecture being considered by the designer. This statement will be further explained in the course of presenting our analysis. We consider two types of architectures. The first type of architecture uses general purpose floating point processors (GPP's) which are capable of executing operations belonging to any of the primitive classes. In contrast, the second type is a VLSI architecture involving specialized computational modules which are dedicated to computations belonging to only one primitive class. We will see how the interpretation of the tabular data varies depending on the type of architecture under consideration.

To process the tabular data within the context of a VLSI architecture, we scan the tabular chart in a column-like fashion for the entire computational cycle looking for the maximum value in each column. This maximum value gives us the number of computational modules of each primitive class that are needed to support all parallel computations. For example, analyzing the data in Table 4-1, we observe that the maximum number of MV and VP computational modules needed to support parallel computations in macrocycles 40 through 44 is three and four, respectively. On the other hand, to process the data within the context of a GPP architecture, we generate a new column in our tabular chart which contains the number of processing elements that are needed in each macrocycle *regardless of what primitive class they belong to*. Then, we search for a maximum value in that new column. This is equivalent to adding all the columns in the tabular chart (i.e., those for each primitive class) and then searching for a maximum throughout the totals obtained for each row. This maximum value gives us the number of parallel GPP's

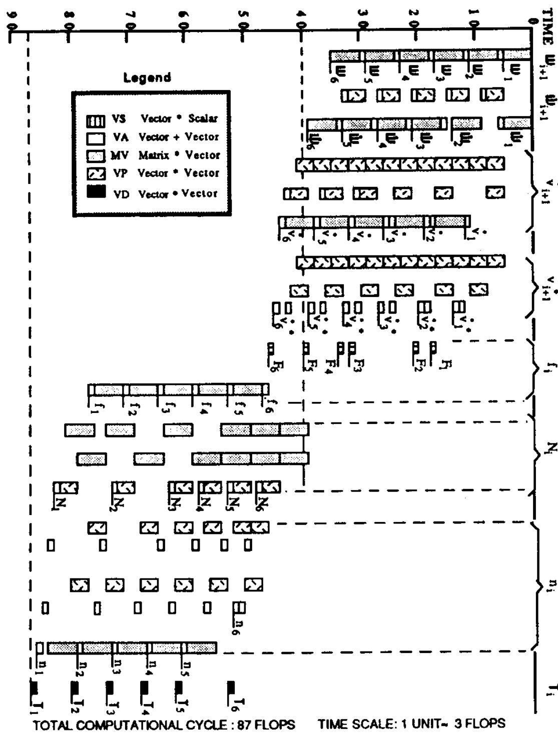

Figure 4-2: Activity Chart for Newton-Euler
Case: N = 6

needed to support all parallel computations. For example, the rightmost column in Table 4-1 depicts the totals obtained by adding the elements for macrocycles 40 through 44. A maximum of seven in the rightmost column indicates that seven parallel processors, at most, are required to support all parallel computational activity in macrocycles 40 through 44. A plot of the values in this new column versus the number of macrocycles is called a *processor activity graph* and is depicted in Figure 4-4. Once tabular charts for each formulation are completed and preliminary worst-case values for the number of processing elements are obtained for each architecture, we minimize the hardware requirements by performing *optimized scheduling.*

	PRIMITIVE CLASSES					
MACROCYCLE	VS	VA	MV	VP	VD	TOTAL
40	0	0	3	4	0	7
41	0	0	3	2	0	5
42	0	2	3	0	0	5
43	0	1	2	0	0	3
44	0	1	2	1	0	4

Table 4-1: Tabular Chart Example for Newton-Euler Illustrating Computational Hardware Requirements for Macrocycles 40-44

Optimized scheduling consists of rescheduling operations taking place in heavily-loaded macrocycles and re-assigning them to macrocycles with a lower utilization of processing elements. The objective is to reduce the preliminary worst-case computational hardware requirements obtained in the second step of the extraction process for both the GPP and VLSI architecture, without increasing the dynamics computational cycle. Simply stated, optimized scheduling is an attempt reduce peak computational demands by attaining a more *even distribution of computational resources.* Once again, this optimization process is dependent on the type of target architecture. For example, in a GPP architecture this optimization process would strive to reduce the total number of processors, whereas in a VLSI architecture the objective would be to reduce the number of computational modules for each primitive class. The result of this step is a scheduling scheme with a minimum number of processing devices and maximum computational speed. In the following Sections the results obtained from the application of the computational extraction process to the N-E and L-E formulations for both, the VLSI and GPP architectures, are presented and compared.

4.1. ANALYZING THE HARDWARE REQUIREMENTS FOR INVERSE DYNAMICS

4.2. Hardware Requirements for Lagrange-Euler

The analysis of the tabular chart for the L-E formulation from the VLSI perspective yields a maximum of 28 computational modules, belonging to the ten primitive classes described in Figure 4-1. The breakdown is shown in Table 4-2. On the other hand, the analysis from the GPP perspective yields a maximum of 11 general-purpose floating point processors. This is the absolute maximum of the processor activity graph for L-E, depicted in Figure 4-3.

PRIMITIVE CLASSES										
MP	SM	MA	MV	VM	VP	TR	AD	VA	SV	TOTAL
10	7	4	1	1	1	1	1	1	1	28

Table 4-2: Primitive Modules needed for a VLSI implementation of the 3x3 Recursive Lagrange-Euler formulation

We compare the results for both architectures according to two important criteria: total number of processing elements and their utilization. First, the L-E formulation requires a relatively high number of primitive classes (ten) which would occupy too much space if they were implemented in integrated circuit form. Second, the tabular chart for L-E indicates that the utilization of primitive modules in a VLSI architecture is very low. For example, in macrocycle 100, as illustrated in Table 4-3, only 5 of the required 28 primitive computational modules will be in use, which corresponds to about 18% of the total available processing power. Unfortunately, this low utilization of computational resources is consistent throughout the L-E computational cycle [15]. These observations indicate that a special-purpose architecture using GPP's would be more efficient than a VLSI implementation. Therefore, we analyze the GPP architecture in more detail in the sequel.

One of the major factors to be taken into account when analyzing the utilization efficiency of a GPP architecture is to observe the processor activity throughout the computational cycle. This task is easily accomplished by analyzing the processor activity graph depicted in Figure 4-3. This graph shows that while the maximum number of GPP's required is 11, the eleven processors will be active only during four macrocycles (numbers 12-14 and number 49). Moreover, a significant decrease in processor utilization occurs during the backward recursions (i.e., for macrocycles > 129), where a maximum of only 3 processors are needed. Therefore, if the L-E formulation is implemented in a special-purpose GPP architecture only 27% of the hardware would actually be used during 52% of the computational cycle. Further, the average processor utilization for the remaining 48% of the computational cycle, corresponding to the forward recursions (i.e., for macrocycles < 129) is also low. These results indicate that a GPP implementation of the L-E formulation also

	PRIMITIVE CLASSES										
MACROCYCLE	MP	SM	MA	MV	VM	VP	TR	AD	VA	SV	TOTAL
100	3	0	0	1	1	0	0	0	0	0	5

Table 4-3: Tabular Chart Example for Lagrange-Euler

results in a very inefficient architecture. In the following Sections we analyze the GPP and VLSI implementation of the N-E algorithm and show that it results in a more efficient architecture.

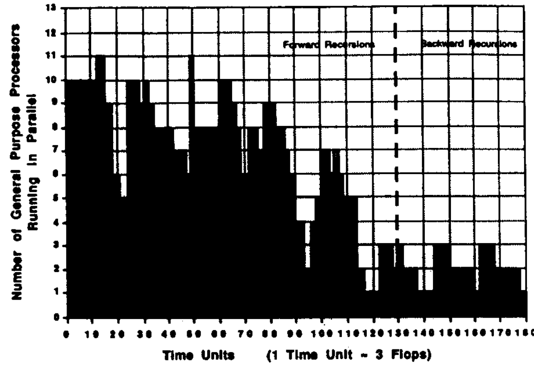

Figure 4-3: Processor Activity Graph for a GPP implementation of Recursive L-E

4.3. Hardware Requirements for Newton-Euler

We analyze the tabular chart for the N-E formulation to determine the computational hardware requirements for a VLSI architecture. We observe that a maximum of 12 computational modules are needed. Each one of these computational modules is capable of executing *one* of the five primitive matrix/vector operations involved in the N-E formulation, described in Figure 4-2. The primitive modules needed for a VLSI implementation is shown in Table 4-4. Before presenting our results for the GPP architecture, we discuss the process of extracting the computational hardware requirements for the N-E formulation.

PRIMITIVE CLASSES					
VD	VC	VS	MV	VA	TOTAL
1	2	3	5	1	12

Table 4-4: Primitive Modules needed for a VLSI implementation of the Newton-Euler formulation

In our analysis, we extracted the computational hardware requirements for the GPP implementation of N-E dynamics by considering two different scheduling schemes, both of which have *identical* computational cycles. The first scheduling scheme includes, in the forward recursions, the execution of primitive computations which are *functionally* associated with backward recursions. In other words, in this scheduling scheme, computational modules are activated as soon as the required operands are available regardless of whether they are partial computations which *functionally* contribute to a forward recursion or backward recursion iteration. On the other hand, the second scheduling scheme attempts to maintain a higher degree of coupling between the primitive computational modules and their functional counterparts (the forward and bacward recursions) by excluding the activation of primitive computations belonging to the backward recursions during the forward recursions. The activity chart depicted in Figure 4-2 corresponds to the second scheduling scheme. In our previous work, we have obtained the activity chart pertaining to the first scheduling scheme [16]. One of the goals of our design methodology is to obtain a high utilization of computational resources. Hence, we must select a scheduling scheme which exhibits a high processor utilization. Due to the higher processor utilization of the second scheduling scheme, which we present in this paper, we have excluded the activity chart for the first scheme in this presentation.

The extraction of computational hardware requirements for the first case produced a processor count of 12, as depicted in Figure 4-4, as opposed to only 8 GPP's for the second case, as illustrated on Figure 4-5). Due to the results obtained, the second scheduling scheme has been selected for a GPP architecture since it has lower computational hardware requirements and also exhibits a more uniform processor utilization throughout the computational cycle. Hence, by scheduling the operations more evenly throughout the dynamics computational cycle, a *decrease* in overall computational hardware requirements and an *increase* in processor utilization is achieved. High-processor utilizations are essential to *special-purpose* architectures, since idle-processing time represents a loss of computational resources. The observations presented in Sections 4.3 and 4.2 are compared in the next Section.

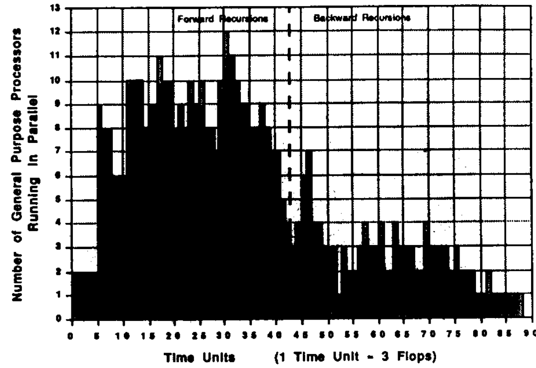

Figure 4-4: Processor Activity Graph for N-E with Partial Computations for the Backward Recursions executed during Forward Recursions

5. COMPARING THE NEWTON-EULER AND LAGRANGE-EULER FORMULATIONS

In this Section, we compare the results obtained for Lagrange-Euler and Newton-Euler in two steps. First, we compare the computational requirements for both formulations. Second, we compare the number of parallel computational units required in each case for both, VLSI and GPP implementations.

To compare the computational requirements for L-E and N-E dynamics, we refer to the activity chart for the 3x3 L-E formulation, depicted in Figure 4-1 and the activity chart for the N-E formulation, depicted in Figure 4-2. Both cases have been illustrated for a 6 DOF rotary manipulator. Several observations are in order: First, the macrocycle lengths for *both cases* are proportional to the time it takes to execute 3 floating-point operations. Therefore, the time scales for both activity charts are expressed in the same units. Second, the computational cycle for the L-E formulation takes 251 macrocycles, or 753 floating-point operations. This represents an 81% reduction in the computational cycle as compared to its sequential implementation which required 3914 flops. By the same token, the computational cycle for the N-E formulation, requiring 87 macrocycles (i.e., 251 flops) represents an 81% reduction as compared to its sequential implementation which requires 1356 flops[7] [16]). Hence, after mathematical decomposition, the length of the L-E computational cycle is about 3 times that of the N-E computational cycle. Before reaching any conlusions about the comparative efficiency of these two formulations we must compare their hardware requirements.

A VLSI implementation of the dynamic equations consists of the interconnection of a group of computational modules belonging to one or more primitive classes. A large variety of primitive classes requires a large area on a silicon chip, which is obviously undesirable. The analysis presented in Section 4.2 demonstrates that the structure of the L-E algorithm is such that it requires a larger number of primitive classes than the N-E formulation (ten primitive classes versus five). Moreover, the excessive amount of computational modules required by L-E (twenty eight) are, in general, underutilized . On the other hand, N-E requires only twelve computational modules which are

[7]Note that the number of flops we have used for a sequential implementation of N-E in this Section is less than the theoretical maximum number of operations presented in Section 2, which are 1590 flops. The difference is due to the application of a few simplifying assumptions we have presented in our previous work.

utilized much more efficiently. Therefore, N-E dynamics is a more desirable formulation for a VLSI implementation.

From the GPP architecture viewpoint, the N-E formulation also leads to a more efficient implementation than the L-E formulation. Our analysis shows that the N-E equations require a maximum of eight GPP's for the parallel architecture, whereas the L-E equations require twelve GPP's. Hence, *not only is N-E about 3 times faster than L-E, but it only requires 67% of the computational hardware required for L-E.*

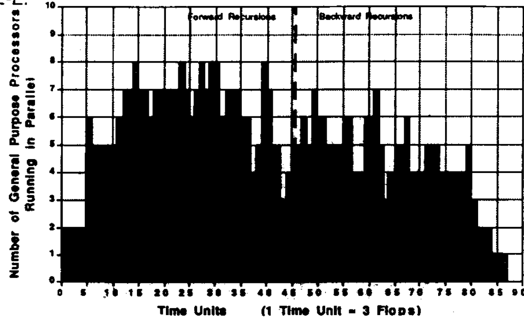

Figure 4-5: Processor Activity Graph for N-E with an even distribution of operations throughout the Computational Cycle

In spite of its disadvantages, the L-E formulation has several advantages over the N-E formulation from a structural point of view. One of the advantages of the L-E formulation is a higher modularity of the computations and fewer inter-chain data dependencies as compared to the N-E formulation. This feature allows us to easily obtain closed-form formulas for the total execution time for a computational cycle as a function of the number of joints [17]. Furthermore, hardware scheduling is more flexible for L-E than for N-E since the computational hardware allocation scheme for L-E is completely independent of the manipulator configuration (i.e., which joints are rotational and which are translational), which is not the case for N-E dynamics. Therefore, once a robot manipulator has been characterized, the same scheduling scheme can be used regardless of any change in its configuration.

Since our main goal is to obtain a high-speed architecture, we conclude that the disadvantages of the recursive L-E algorithm outweigh its advantages. Therefore, due to the increased speed, reduced computational hardware requirements and higher utilization of computational resources, we propose the Newton-Euler formulation as the basis of a multiprocessor implementation of inverse dynamics.

The actual design of a multiprocessor implementation of the N-E formulation requires a careful evaluation of a number of tradeoffs such as: internal buffering and storing of intermediate results, storing and accessing global variables with deadlock prevention, and inter-processor communication mechanisms. All of these factors should be considered in the design of an efficient memory configuration scheme and a compatible data allocation mechanism. In addition, the design of a memory configuration scheme involves a careful analysis of what impact the use of local or global memories will have on the performance of the parallel architecture. A presentation of our architectural design is beyond the scope of this paper; however, we include a detailed description of our analysis in [15].

6. SUMMARY

In this paper, we have presented a new approach for mapping parallel algorithms into architectures. We have demonstrated the use of our methodolgy by applying it to both the L-E and N-E manipulator dynamics formulations. Our scheme for scheduling the parallel computations is called the *mathematical decomposition* scheme. We have used this decomposition scheme to obtain the hardware requirements for both the Newton-Euler and Lagrange-Euler formulations. We have also compared the hardware requirements for both of these formulations, as well as some of their structural features, in order to propose an efficient formulation to be used as the basis for the implementation of a special-purpose dynamics computation architecture. Our results show that the Newton-Euler formulation is about three times faster than the Lagrange-Euler formulation and requires less hardware to compute the dynamics equations in real-time. In our analysis of special-purpose parallel architectures, we have considered the implementation of two architectures : a VLSI architecture and a multiprocessor architecture consisting of general purpose floating point processors (GPP's). Our analysis shows that the GPP architecture has less hardware requirements compared to the VLSI architecture and is better suited for a practical implementation.

References

[1] John M. Hollerbach.
A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation Complexity.
IEEE Trans. on Systems, Man, and Cybernetics SMC-10(1):730-736, November, 1980.

[2] Hollerbach,J.M. and Sahar,G.
Wrist Partitioned Inverse Kinematic Accelerations and Manipulator Dynamics.
In Paul, R.P. (editor), *Proceedings of the first International IEEE Conf. on Robotics and Automation*, pages 152-161. IEEE, March, 1984.

[3] Kanade, T., Khosla,P.K. and Tanaka,N.
Real-Time Control of the CMU Direct Drive Arm II Using Customized Inverse Dynamics.
In Polis,M.P. (editor), *Proceedings of the 23rd IEEE Conference on Decision and Control*, pages 1345-1352. IEEE, Las Vegas,NV, December 12-14, 1984.

[4] Khosla,P.K and Kanade, T.
Real-Time Implementation and Evaluation of Model-Based Controls on CMU DD ARM II.
In Bejczy,A.K. (editor), *1986 IEEE International Conference on Robotics and Automation.* IEEE, April 7-10, 1986.

[5] Khosla, P.K. and Neuman, C.P.
Computational Requirements of Customized Newton-Euler Algorithms.
Journal of Robotic Systems 2(3):309-327, Fall, 1985.

[6] P.M. Kogge and H.S. Stone.
A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations.
IEEE Trans. on Comput. C-22:789-793, Aug., 1973.

[7] Richard H. Lathrop.
Parallelism in Manipulator Dynamics.
Intl. Journal of Robotics Research 4, 1985.

[8] C.S.G. Lee, T.N. Mudge, and J.L.Turney.
Hierarchical Control Structure Using Special Purpose Processors for the Control of Robot Arms.
In *Proceedings 1982 Pattern Recognition and Image Processing Conf.*, pages 634-640. Las Vegas, Nevada, June 14-17, 1982.

[9] Luh, J.Y.S., M.W. Walker, and R.P.C. Paul.
On-line Computational Scheme for Mechanical Manipulators.
Trans. of ASME, J. of Dynam. Syst.,Meas., and Contrl. 102:69-76, June, 1980.

[10] Luh, J.Y.S. and C.S. Lin.
Scheduling of Parallel Computation for a Computer-Controlled Mechanical Manipulator.
IEEE Trans. Syst. Man, and Cyber. SMC-12(2):214-234, March/April, 1982.

[11] Nigam,R. and Lee,C.S.G.
A Multiprocessor-Based Controller for the Control of Mechanical Manipulators.
IEEE Journal of Robotics and Automation 3(1):214-234, April, 1986.

[12] Orin,D.D.
Systolic Architectures for Computing Structures for Robotics Applications.
Specialized Computer Architectures for Robotics and Automation.
Gordon and Breach Publishers, New York, to appear.
Edited by James Graham.

[13] Raibert,M.H.
Analytical Equations vs. Table Lookup for Manipulation: A Unifying Concept.
In *Proceedings of the IEEE Conference on Decision and Control*, pages 576-579. IEEE, New Orleans, La., December, 1977.

[14] Raibert,M.H. and Horn,B.K.P.
Manipulator Control Using Configuration Space Method.
Industrial Robot 5:69-73, June, 1978.

[15] Sandra Ramos.
A High-Speed Parallel Architecture for Robot Control.
Master's thesis, Carnegie Mellon University, To appear, 1988.

[16] Ramos,S. and Khosla,P.K.
A Computational Decomposition Scheme for the Newton-Euler Dynamic Formulation.
Technical Report, Carnegie-Mellon University, to appear.

[17] Ramos,S. and Khosla,P.K.
A Parallel Computational Scheme for Lagrange-Euler Dynamics.
30th Midwest Symposium on Circuits and Systems , August, 1987.