Evaluating Multigauge Architectures for Computer Vision

by
W. B. Ligon III and U. Ramachandran

TR-111392-0915P
Electrical and Computer Engineering Department
Clemson University
Clemson, SC 29634-0915

Evaluating Multigauge Architectures for Computer Vision *

W. B. Ligon III
Dept. of Elec. and Comp. Engr.
Clemson University
Clemson, SC 29634-0915
Ph: (803) 656-1224
Fax: (803) 656-5910
email: walt@eng.clemson.edu

U. Ramachandran
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Ph: (404) 853-9368
Fax: (404) 853-9378
email: rama@cc.gatech.edu

Abstract

Heterogeneous computing is a technique of achieving high performance by providing a variety of different architectures to meet the needs of systems that are composed of tasks with widely different characteristics. Essential to the construction of heterogeneous systems is an understanding of the match between architecture and software and how that match can be used in deciding how to utilize the available computing resources. We present a theoretical framework, the PCI Model, which defines corresponding characteristics of parallel programs and parallel architectures and defines the performance relationship between them in terms of these characteristics. We have encapsulated the concepts of the PCI model into RAW, a simulation environment that facilitates experimentation with the program/architecture relationship in terms of the PCI model. Using RAW, we have applied the PCI model to study the use of processor econfigurable architectures (a type of heterogeneous system) in the context of computer vision applications. We present experimental results that demonstrate that these architectures perform better than static homogeneous architectures for such applications.

Keywords

heterogeneous computing, parallel processing, multigauge architectures, reconfigurable architectures, execution-driven simulation, computer vision.

*This work is funded in part by an NSF PYI award MIP-9058430.

1 Introduction

The computing community is increasingly turning to parallel processing as a means of improving the performance of compute intensive applications. Over time, research has yielded a number of very different parallel processing architectures designed to provide this high performance. At the same time, the complexity of the applications has increased, and it has become clear that many compute intensive problems are composed of many phases or *tasks* each of which may perform very differently on the various architecture available. Thus, today we are faced with a multiplicity of architectures, each of which performs very well for some tasks, and not as good for others. In order to address this situation, research is being conducted on the use of heterogeneous parallel computing systems. A heterogeneous computing system provides multiple architectures so that the needs of each task of a program can be met to the fullest extent possible. This requires decomposing the target program into its component tasks, assigning each task to an architecture, parallelizing the task to run on that architecture, and providing the necessary coordination to deal with the partitioned program.

There are many approaches to constructing heterogeneous systems. One approach is to provide several different kinds of architectures connected via a network, thus allowing different tasks to be run in parallel on the various architectures provided. This approach is appealing because it is relatively simple to gather together the various types of hardware using currently available commercial computers. Another approach is to develop new parallel architectures that provide multiple processor architectures and/or multiple models of computation. The appeal to this approach is that there is greater flexibility in providing a mix of resources that best suit the target application. Finally, a third approach to the development of heterogeneous computing systems is the use of reconfigurable architectures. These architectures can dynamically alter their logical structure to provide the desired mix of computational resources and computing modes. This approach provides the greatest flexibility in resource allocation because the allocation is made at run time to best suit the current state of the application. This flexibility is at a cost, because custom hardware is required to achieve reconfiguration.

Regardless of the approach used to develop heterogeneous systems, there are a number of issues one must address in their design. Ideally, for each task in the target application, one would like to know the best architecture possible for that task. Next, one needs to decide how to interconnect the architectures in order to minimize the costs of coordinating the system such as communication and synchronization delays. In the real world there are many problems with these issues. First, our ability to determine the best architecture for a task is ad hoc at best. Second, it is often not cost effective to construct a brand new architecture that is an exact match for each task. In fact, one may be faced with fitting a task to the best candidate among several existing

architectures. Thus, it is not enough to know the best architecture for a task; rather, one needs a quantitative measure of each task's expected performance for any given architecture as compared to another. Next, on the issue of coordination costs, again one rarely has the liberty of constructing the interconnect to exactly match the needs of the application. Rather, one usually has to contend with existing facilities, so again there is a need for a quantitative understanding of the cost of communication and synchronization between the components of the system so one can determine if those costs are outweighed by the benefit provided in partitioning the tasks to different architectures. Reconfigurable architectures appear to avoid these issues by providing enough flexibility to adapt to each task's requirements. Again, in the real world this turns out to be only a matter of degree. Reconfigurable architectures are not infinitely flexible, so the cost/benefit tradeoff continues to exist. In fact, there is the added problem that since the same resources are configured to achieve the various architectures, one must consider which task benefits the most from the resources available.

In summary, the successful development of heterogeneous parallel computing systems depends on our ability to quantify the fit of a program task to a given architecture and to quantify the costs involved in the interaction between the architectures. Unfortunately, we do not currently possess a rigorous means of addressing these issues. Rather, the prevailing approach is based on ad hoc techniques, trial and error, and intuition. The goal of our research is to begin to address this problem by establishing a model for reasoning about the performance of parallel programs relative to a specific parallel architecture. Our initial effort has been to develop a simulation testbed based on this model with which we can empirically explore the program/architecture relationship. Ultimately, we hope that through these explorations we will extract characteristics of programs and architectures that will enable us to develop analytical techniques for predicting program performance.

In this paper we present the *Processor*, *Control*, *Interconnection* (PCI) model for programs and architectures. The PCI model separates processor design, control structure, and interconnection issues and also includes a means for dealing with architectural reconfigurability. We further discuss the performance relationship of programs and architectures within the context of this framework. These concepts have been embodied in a set of tools, the Reconfigurable Architecture Workbench (RAW), which allow us to simulate the execution of parallel programs on models of various parallel architectures. In [7] we discuss the design details of RAW and provide an example experiment using it. In this paper we focus on the PCI model and present the results of experiments utilizing a coordinated set of tasks from computer vision applications. In [8] and [5] we demonstrate the PCI model's ability to handle control and interconnection aspects of the architecture. The experiments in this paper focus on processor granularity as the experimental variable, thus this paper emphasizes a breadth of application tasks rather than a breadth of architectural issues. Our purpose is to demonstrate the PCI model's utility in exploring

the performance characteristics of complex programs in a heterogeneous environment through the specific results we present. Section 2 presents the details of the PCI model; section 3 is brief introduction to the RAW toolset; and section 4 presents the programs and architectures used in the experiments and discusses the results obtained.

2 The PCI Model

The purpose of the PCI model is to provide a framework for reasoning about the performance relationship of programs and architectures. To do this, the model must include a means for describing the structure of parallel programs and the structure of parallel architectures and a means for determining the performance implications of the relationship between these structures. This information can then be used to guide the development of analysis tools and techniques. An important feature of the PCI model is that each aspect of the model is based on a common set of three dimensions. This organization allows us to separate out issues related to each dimension so that we can study the implications of those issues in isolation.

The PCI model defines parallel programs as being composed of three components: 1) independent *instruction streams* each of which execute in a sequential manner, 2) independent *data streams*, each of which execute instructions provided by an instruction stream on different data elements, and 3) *communication* between the data streams and synchronization between the instruction streams which may take place either by message passing or via shared memory. PCI defines parallel architectures as being composed of three similar components: 1) *control units* (CUs) that process instruction streams, 2) *processing elements* (PEs) that execute the instructions issued by a given control unit, and 3) *interconnection networks* (ICNs) that allow synchronization among CUs and exchange of data among PEs (and their memories). We call these components the control configuration, the processor configuration, and the interconnection configuration. Finally, PCI defines three classes of reconfigurability: 1) processor reconfiguration allows the architecture to trade off the number of PEs for the precision and speed of the PEs, 2) control reconfiguration allows the assignment of PEs to control units to be changed, and 3) interconnection reconfiguration allows the communication capabilities of the parallel architecture to be modified.

2.1 The PCI Model for Parallel Programs

The class of parallel programs we are studying includes those that consist of a set of data objects and a set of instructions for manipulating those objects. The instructions for the program may include communication instructions which serve to transfer information from one context to another. In addition, most programs include input/output, which can be considered as a special case of communication. When a program is executed, it produces one or more instruction streams which are loosely defined as sequences of instructions issued when

the program is executed. If a program's execution yields more than one instruction stream, the instructions of the various instruction streams may be issued simultaneously (parallel execution), or they may be interleaved in time (concurrent execution). An instruction stream I is defined as a sequence of instructions $\{i_0, i_1, i_2, \ldots, i_n\}$ such that the issue of instruction i_j implies the issue of i_{j+1} as the next instruction in the instruction stream in all correct interleavings of the program's instruction streams, including parallel execution. Programs that define multiple instruction streams are termed *function parallel* or *control parallel* programs.

Each instruction of an instruction stream carries out a specific manipulation on one or more data objects defined by the parallel program. *Data parallel* programs are those that define multiple data objects that may be manipulated simultaneously by the instructions of a single instruction stream. In order to perform this task, the program must define a parallel data space PD, composed of n constituent spaces pd_i. Parallel data objects PO consists of n constituent objects po_i each of which exists in the corresponding data space pd_i (see Figure 1). Similarly, there is a single data space SD where singular data objects are defined. This data space is further

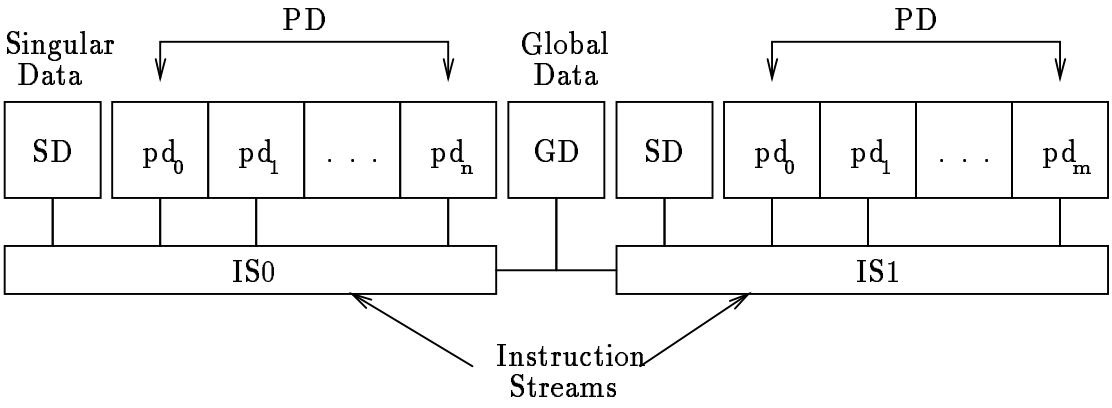

Figure 1: PCI Program Model.

partitioned into a local data space, private to a given instruction stream, and a global data space GD, shared by all instruction streams. A data stream is defined as a sequence of parallel objects $\{PO0, PO1, PO2, \ldots, POn\}$ from a single constituent of the parallel data space pd_i which are manipulated by an instruction stream. Thus, programs that define parallel data objects and manipulate them define multiple data streams.

Communication is the process of transferring information from one data space to another. Communication within an instruction stream is essentially data transfer between parallel data spaces. Communication between different instruction streams may be data transfers or transfers of control information as in synchronization.

Communication generally takes one of two forms: sending and receiving messages, and access to data objects in a shared data space.

There exists a special class of instructions known as `WAIT` instructions. A `WAIT` instruction is an instruction that does not complete until some specified event has occurred. These events are usually the result of some type of communication. For example, an instruction stream may issue a *blocking receive* instruction (one type of `WAIT`) which does not complete until a message is received from some other instruction stream. Another important example is a *memory load* instruction which does not complete until the memory value is returned. Different systems may implement a variety of `WAIT` instructions, but in the final analysis, the effect of these instructions is that a delay is introduced into the instruction stream's execution time which may be dependent on events that occur in other instruction streams.

2.2 The PCI Model for Parallel Architectures

In the PCI model for parallel architectures (see Figure 2), we take the classical notion of a processor and divide it into a control unit and a processing element. These are combined with various memories and interconnection channels to form the class of architectures we study. Control units (CUs) are devices that process instruction

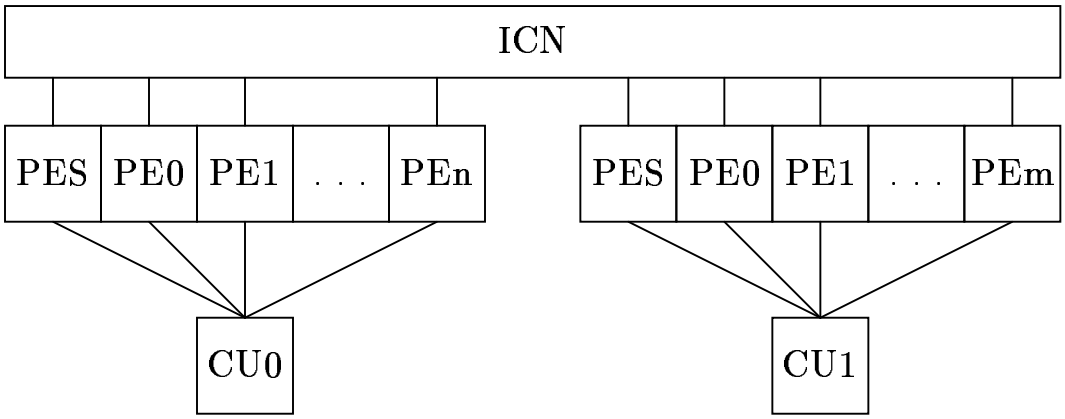

Figure 2: PCI Architecture Model.

streams. A CU consists of a program counter, instruction memory, and whatever logic is needed to generate the signals needed by the processing elements to carry out the instructions of the program. Branching instructions are carried out directly by the CUs, while all other instructions are handled by PEs. Each CU in a system is capable of generating a single instruction address during each machine cycle. Each instruction address may

cause one or more instructions to be fetched from the instruction memory. A single instruction man be issued in parallel to all PEs controlled by the CU (SIMD processing), or the CU may issue different instructions to each PE (VLIW processing). CUs may contain additional features that support the processing of multiple concurrent instruction streams, but only one instruction stream can be active during any one CU machine cycle.

Processing elements are devices that execute instructions by manipulating data objects. PEs consist of execution logic, operand address generation logic, and memories that define the various data spaces. In this way, the PEs support the processing of data streams. There always exists at least one PE for each control unit called the *singular* PE (labeled "PES" in the figure). This PE's memory defines the local data space and its execution logic carries out the instructions directed at the objects in that data space and the objects in the global data space. In addition to the singular PE, each control unit may have $0 \ldots n$ *parallel* PEs (labeled "PE0" to "PE7" in the figure) which define and operate on the objects in the parallel data spaces.

A PE is characterized by its *word width* and *instruction timings*, which are the number of machine cycles required to execute each of the instructions that might be issued to it by the control unit. The word width defines the largest data object that can be processed in a single machine cycle by the PE, and the instruction timings depend on the complexity of the computational hardware provided.

2.2.1 Interconnection Networks

Interconnection Networks consist of two basic elements: *links* and *switches*. A link is a medium that is used to pass information from one node to another node or group of nodes. A *node* can either be a *terminal*, or a *switch*: A terminal is any active architectural element such as a processor, a memory module, an I/O device, or a controller. A switch is an active device in a network that serves to pass information from the end of one link to the beginning of another. Various switches may have differing amounts of local intelligence. A switch may make routing decisions, it may store messages for later transmission, and it may choose to allocate or deallocate network resources. The links and switches of an ICN are characterized by a set of parameters:

- *Topology* describes the collection of links and switches in the ICN and their interconnection.
- *Bandwidth* describes the throughput of the links and switches in bits per unit time.
- *Latency* describes the delay incurred through a link or a switch by a flit [1].
- *Switch capabilities* describe the level of intelligence incorporated into a switch, such as routing logic, queueing, and policies.

[1] A *flit* is the number of bits that can be transmitted in a single network clock cycle

2.2.2 Architectural Reconfigurability

In order to understand architectural reconfigurability, one must first distinguish between the physical and logical view of a given architectural element. Architectural reconfiguration is the process of altering the logical view of the machine in order to give the appearance of a different architecture. This process is typically performed in the physical view by establishing communication between physical components of the machine and utilizing those components as if they were a single architectural entity. The cost of this added flexibility is a possible reduction in efficiency due to the inability to optimize for a single logical organization. There are three types of reconfigurability:

Interconnection reconfigurability describes the flexibility of an ICN in allocating its resources to provide service between two terminals. Non-reconfigurable ICNs have only a fixed set of resources for building a path between terminals. A reconfigurable network allocates switches and links from a pool to build a path between terminals. Essentially, reconfigurability is a means of providing a high degree of service for short periods of time over a subset of the system with fewer total resources. An example of an interconnection reconfigurable architecture is the CHiP architecture [12].

Control reconfigurability (also known as multi-mode capability) describes the ability of an architecture to partition the PEs in the system among the CUs in various ways. Control reconfigurable machines can be configured as MIMD, SIMD, and Multiple-SIMD systems. Examples of systems which perform control reconfiguration are the Connection Machine 2 [3] which provides 64K PEs and 4 CUs, TRAC [10],and PASM [11], each of which provide N PEs and CUs.

Processor reconfigurability (also known as multi-gauge capability) is the ability of the architecture to trade off faster and/or higher precision PEs for more numerous PEs. This capability can take two forms. First, low-precision PEs can be logically joined to form higher-precision PEs. This type of processor reconfiguration is called *precision reconfiguration* and has been provided in TRAC and DCG [4]. Second, simple boolean logic units can be combined to form more complex arithmetic units such as fast multipliers and floating point units. This feature is called *capability reconfiguration* and is introduced in [6]. Capability reconfiguration is radically different from precision reconfiguration in terms of its implementation requirements. Capability reconfigurable PEs begin with a very simple computation circuit such as that described in [3], but with a full word of precision. In such systems the CU utilizes a microsequencer to decode complex arithmetic operations such as multiplication and floating point arithmetic into a sequence of microinstructions. In order to perform capability reconfiguration on the PEs, one needs to execute a control-parallel microprogram on interleaved sets of the simple PEs. For

example, a multiplier can be constructed by using one simple PE as a shifter, and an adjacent simple PE as an adder to perform shift-add multiplication considerably faster than using one simple PE. To implement this capability, the CU must provide multiple microinstruction busses to the PEs, and multiple microinstruction stores, one for each bus (see Figure 3). Routines for simple PEs have the same code loaded at corresponding

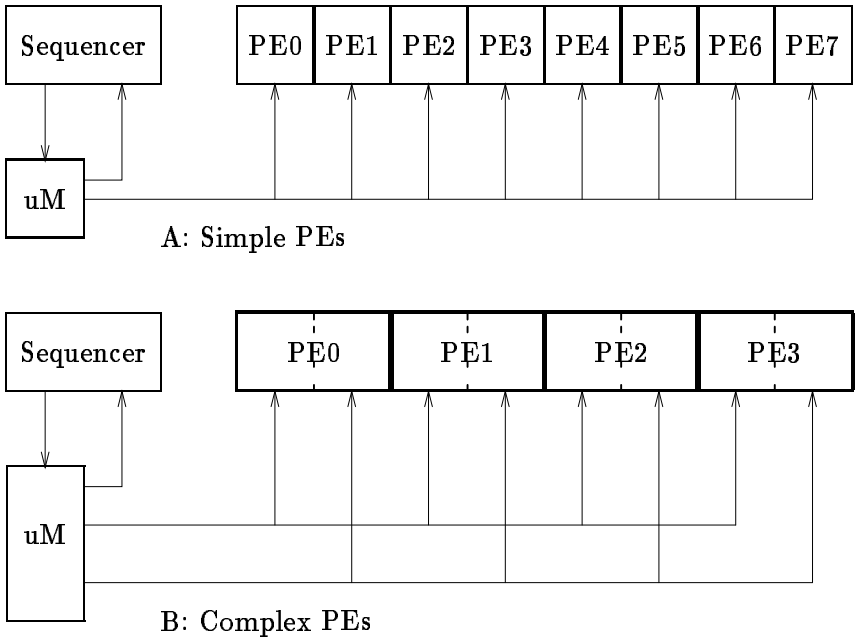

Figure 3: Capability Reconfiguration.

addresses in the various microstores. Routines for the complex PEs have different code loaded at the same address. In addition, there must exist some means for passing data between the simple PEs and performing basic coordination tasks.

2.3 Understanding the Performance of Programs Relative to Architectures

We now consider how the relationship between these models results in a given level of machine performance. Our measure of performance is the execution time of the program on a given architecture. The execution time of a program on a particular architecture can be found by considering the instructions issued by each CU. The execution time of m instruction streams running concurrently on a single CU is:

$$T_{execution}(P) = \sum_{k=0}^{m} \sum_{j=0}^{n(k)} t_{instruction}(i_{j(k)})$$

where $n(k)$ is the number of instructions in instruction stream I_k. The function $t_{instruction}((i_{j(k)})$ is the instruction time which is defined as

$$t_{instruction}(i_{j(k)}) = t_{issue}(i_{next}) - t_{issue}(i_{j(k)})$$

where $i_{j(k)}$ is instruction i_j of instruction stream I_k and i_{next} is the next instruction issued by the CU that instruction stream I_k is executing on. In the case where I_k is the only instruction stream executing on the CU, i_{next} is guaranteed to be $i_{j(k)+1}$, but this may or may not be the case otherwise.

In the case where a program's m instruction streams are executing in parallel on the target architecture, each on a different CU the execution time of the program is the same as that of the longest instruction stream:

$$T_{execution}(P) = \max(T_{execution}(I_k))k = 0 \ldots m.$$

Instruction streams that do not start at time t_0 are modeled as an instruction stream that does begin at time t_0 and executes a `WAIT` instruction, which causes the instruction stream to wait until its actual execution time.

2.3.1 The Effect of Control Units

The effect of control units on program performance depends first on the number of instruction streams in the program. Control units in excess of the number of instruction streams have no effect on the program's performance whatsoever. If a program has multiple instruction streams and they all execute concurrently on a single CU, the CU can process instructions from other instruction streams during the time one instruction stream is waiting, thus `WAIT` instructions have relatively little effect on overall execution time of the program. As more and more CUs are made available to process instruction streams in parallel, the execution time of the program is generally reduced because each CU has fewer instructions to process. However, as more parallel execution is employed, the instruction time of the `WAIT` instructions may increase. If this occurs, then the benefits of parallel execution are diminished. Considering the effect of taking a program P with two instruction streams I_0 and I_1. Assume the execution time of P using one CU is $T_{sequential}(P)$ and the contribution to the execution time of P by I_0 is $T_{sequential}(I_0)$ and similarly $T_{sequential}(I_1)$ for I_1. Now, the execution time of I_0 and I_1 using two CUs is

$$T_{parallel}(I_0) = T_{sequential}(I_0) + \Delta T_{\texttt{WAIT}}(I_0)$$

$$T_{parallel}(I_1) = T_{sequential}(I_1) + \Delta T_{\texttt{WAIT}}(I_1)$$

and the execution time of P is

$$T_{parallel}(P) = \max\left(T_{parallel}(I_0), T_{parallel}(I_1)\right).$$

$T_{\texttt{WAIT}}(I_k)$ is the sum of the execution time of all WAIT instructions $t(i_W)$ each of which can be decomposed into the sum of two values: $t_{sync}(i_W)$ and $t_{comm}(i_W)$. The synchronization time of a WAIT instruction ($t_{sync}(i_W)$) is the difference between the time of the actual occurrence of the event the instruction is waiting for and the issue time of the WAIT instruction, which is due primarily to the program's implementation and the speed of the PEs. t_{sync} is an abstract value that may be positive or negative, depending on whether an event occurs before or after the issue of the WAIT instruction. The communication time of a WAIT instruction ($t_{comm}(i_W)$) is the delay between the actual occurrence of an event and the time the CU receives notice of the event, which is primarily due to the message latency of the intervening ICN and system overheads. t_{comm} is a physical value and thus is strictly non-negative. With these definitions we can find the delay introduced by a WAIT instruction as:

$$t(i_W) = \max(t_{sync}(i_W) + t_{comm}(i_W), 0)$$

which is also strictly non-negative.

When there are many instructions streams executing together on a single CU, the effect of WAIT instructions is minimized by delaying their issue and processing other instruction streams instead. Ideally, this creates negative synchronization time and thus the WAIT instructions have little effect. During parallel execution, $T_{\texttt{WAIT}}(I)$ may increase relative to the interleaved execution for two reasons. First, even though there may still be multiple instruction streams on a CU, there may be times when all of them need to issue a WAIT and thus there are no other instructions to issue. Second, by definition the fact that the event of interest may be occurring on a remote CU, an increase in t_{comm} may be experienced. Thus, the effect of CUs on program performance is paramount and can only be determined by considering the particulars of the target program.

2.3.2 The Effect of Processing Elements

Processing elements are central to the understanding of a program's execution time for it is the time required by the processing elements to perform the manipulations specified by the instructions that usually accounts for the bulk of the execution time. Two issues are at work. The first is simply the time it takes to execute each instruction. The second is the number of PEs available to process parallel data streams that exist in the program. Basically, if there are d data streams and p PEs, then the execution time for a given instruction is

$$T_{execution} = \lceil d/p \rceil .$$

Given this relationship, one naturally concludes that faster PEs produce a faster machine, and more PEs produce a faster machine until there are more PEs than data streams. The problem is, in designing an architecture, one has to contend with several real-life limits such as the amount of hardware than can be made available

as PEs based on an acceptable cost for the machine; physical constraints such power, cooling, size, and signal propagation delays. Given these limits, one must choose the best trade-off between the number of PEs and the speed of the PEs.

2.3.3 The Effect of Interconnection Networks

As discussed in section 2.1, communication in a parallel program can be between instruction streams, or between data streams in a single instruction stream. The effect of ICNs on program performance can be seen in the communication component of the `WAIT` instruction. When communication occurs between data streams in a single instruction stream, the delay of the associated `WAIT` instruction is due entirely to communication time. In this situation, the event the program waits on is the sending of data, which occurs no later than the issue of the `WAIT` instruction. When communication occurs between instruction streams, the `WAIT` instruction delay is due to both the communication time and the synchronization time if the data is sent after the `WAIT` instruction is issued.

The quantity of interest in measuring communication time is message latency which is the difference in time between the sending of the first quantum of data in the message and the receiving of the last quantum of data in the message. This quantity depends on the parameters of the ICN. The relationship between these parameters is beyond the scope of this paper, and the reader is referred to texts such as [2, 1, 9]. One should note that one potentially large component of this equation is that of resource contention which can be highly dependent on the behavior of the parallel program.

2.3.4 The Effect of Architectural Reconfiguration

Architectural reconfiguration differs from all other architectural features in the way it affects program performance in that reconfiguration draws its benefits in exploiting the dynamic changes in parallel program characteristics. The basic premise is this: if one part of a program performs better on architecture X than architecture Y, and a different part of the same program performs better on architecture Y than architecture X, then the program should perform better on an architecture that is capable of reconfiguring both as X and Y than on either X or Y alone. To understand the potential for architectural reconfiguration, then, one must ask this question: Why would one program perform better on one architecture than another? The reasons why a program would perform on one architecture than another are 1) The program cannot utilize the resources provided by the architecture as they are logically configured; 2) a different logical configuration of the architecture's resources would provide improved economies of scale; and 3) utilization of the architecture's resources as logically configured results in a

greater increase in WAIT time than decrease in instruction execution time.

The first problem is easy to understand. If the architecture provides more PEs than the program has data streams or more CUs than the program has instruction streams, then clearly resources remain idle. By a similar argument, if the architecture defines special purpose hardware such as a floating point unit, and the computation is primarily integer, then again, those resources cannot be utilized. The second problem is more subtle. More complex computational units are, at least in theory, less efficient than simple computational units because there is overhead involved in coordinating the added complexity. For example, an n-bit PE is not necessarily n times faster than a 1-bit PE, in part because the carry computation increases the critical path of the adder circuits. In practice, the exact performance relationship between various designs is a complex issue that is beyond the scope of this paper. Furthermore, it is not a goal of this research to outline and classify these issues. Rather, we note that each specific hardware configuration has its own performance characteristics, and that a comparison between then usually involves an empirical analysis of the cost/benefit function. It is this analysis that concerns us. We further note that the goal of a reconfigurable architecture is to provide a means for managing these overheads to yield the most efficient computational engine over a range of possible situations. Finally, the third problem relates program behavior to synchronization cost in the system. As we have seen in section 2.3.1, certain programs can incur a significant increase in the synchronization component of the WAIT instructions if there is not a good match between the control configuration of the architecture and control parallelism in the program.

In summary, the PCI Model defines three dimensions of parallel programs and parallel architectures. These dimensions are useful in that they define the classes of programs and the classes of architectures we are studying and give us a set of concepts and terminology with which to reason about the nature of these two entities and their relationship with respect to performance. This framework gives us a means of evaluating the performance trade-offs available in designing systems for a specific application. In particular, this framework gives a tool for evaluating heterogeneous processing techniques by helping us to understand what parts of the program are best suited to which type of architecture and quantifying the benefits to be had in improving the fit. In order to facilitate the exploration of the program/architecture relationship using the PCI model, we have embedded its concepts into RAW, a set of tools that allows us to analyze a given program/architecture pair. In the next section we briefly describe the RAW tools, then in section 4 we present results obtained in applying them.

3 The Reconfigurable Architecture Workbench

In section 2 we present the PCI model for parallel programs and parallel architectures and we develop a technique for determining the performance of a program on an architecture that utilizes such models. We have designed and implemented an execution-driven simulation environment that embodies these techniques to facilitate experimentation with programs and architectures. Figure 4 depicts the major components of RAW. Programs are translated into *Architecture Independent Code* (AIC) which is based on the PCI model for programs. Architecture

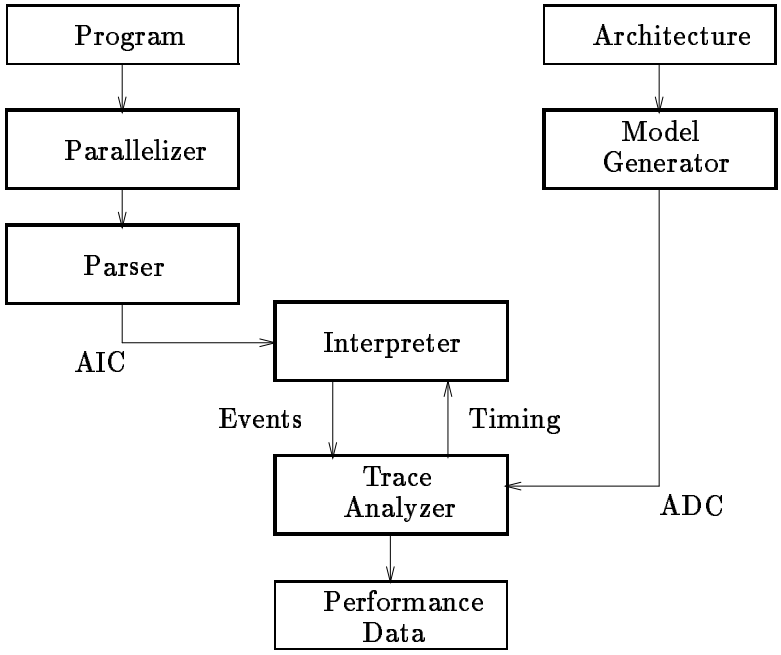

Figure 4: The Reconfigurable Architecture Workbench.

models are encoded into *Architecture Description Code* (ADC) which is based on the PCI model for architectures. The Interpreter combines the AIC and ADC to produce an architecture dependent execution of the program, and the Trace Analyzer utilizes the analysis in section 2.3 to simulate the program's performance and gather performance statistics. Each of these components can be further divided into three components which represent the processor, control, and interconnection dimensions. Details of RAW's design and construction are presented in [7] with additional details given in [5].

4 A Study of Multigauge Architectures

In this section we utilize the PCI model to study the potential for performance gains to be found in using multigauge heterogeneous architectures for computer vision systems by experimenting with the RAW simulation environment. We have selected computer vision as our target application area because there is a significant amount of parallelism and intuitively, the applications lend themselves to the use of heterogeneous architectures. This intuition is gleaned from the observation that vision systems span several levels of abstraction, each of which typically requires different data representations and algorithms for manipulating the objects at that level. Table 1 summarizes the architecture of computer vision systems with respect to the complexity of the computations and quantity of data. The lowest level of abstraction operates on raw image data that might be generated by

	Data Granularity	
Level	Complexity	Quantity
Low	simple	$\approx 1M$
Middle	moderate	$\approx 10K$
High	complex	≈ 100

Table 1: Levels of abstraction in computer vision systems versus data granularity.

a number of different sensor types (visual spectrum camera, infrared spectrum camera, sonic depth finder, laser depth finder, etc.) and typically includes traditional digital image and signal processing algorithms intended to reduce noise, enhance detail, perform segmentation, etc. Data representations at this level are generally quite simple, on the order of 1-5 values per object, and the quantity of data is quite large, on the order of 1 million objects per image. The middle level of abstraction attempts to single out pixels in the raw image data by identifying primitive image features such as edges, textures, regions, corners, etc. Next, an attempt is made to group these primitive features in order to identify more complex features such as lines, arcs, polygons, etc. The data representation at this level is somewhat more complex, as it may involve arbitrary lists and hierarchies of primitive features. An interesting aspect of this level of abstraction is that it naturally reduces the number of data objects by combining many primitive features into fewer complex features. At the highest level of abstraction, computer vision systems must manage a database of objects found and a complex set of *a priori* information that is used in identifying the objects seen in the images, and directing further processing of the images. Typical data might include current environmental conditions and models of expected objects. These objects involve not only arbitrary data structures, but also multiple sets of these data structures. The number of data objects at this level is relatively small.

In order to capture the multi-level aspects of computer vision systems, we have utilized the 2nd DARPA image understanding benchmark in this study [13]. The benchmark uses two input images: an intensity image and a depth image, and is designed to require the use of both bottom-up and top-down strategies to resolve ambiguities in these images. The overall task of the benchmark programs is to recognize objects in a simplified blocks world. In other words all objects to be recognized are composed of a collection of rectangles of various sizes, shapes, orientations, and colors defined in 2 1/2 dimensions. The input images include noise and "false" rectangles that cause ambiguities in the images that can only be resolved by utilizing all of the techniques present in the benchmark. The rectangles found in the images are compared against a database of object models and the best match is identified as the viewed object. The benchmark programs presented in this study are shown in Table 2.1. Implementation details of these programs is beyond the scope of this paper. See Weems [13] for a functional description of the benchmark and [5] for a detailed description of the parallel implementation used in this study.

4.1 Architectures Studied

The models and techniques presented in section 2 can be used to study heterogeneous systems regardless of the approach used to realize these systems. In the study presented here, we restrict ourselves to the use of reconfigurable architectures, and in particular, to the use of multigauge architectures. We have done this for two reasons. The primary reason is to restrict the domain of our analysis so that we may be able to present more detailed results for that domain. Second, reconfigurable architectures provide the most flexible form of heterogeneity (though at a considerable expense), thus they should be able to take the best advantage of any available performance advantage. Once this baseline is understood, we can consider what degradation (if any) less flexible though less costly systems, such as a set of networked multi-granular parallel machines, may afford us.

There are five different PE models used in the course of our experiments. Two of these, the *unit* and *warp* PE models are default models and are intended to serve special purposes. The unit PE model defines all instructions in the AIC instruction set as requiring 1 cycle to complete. This model has a very low simulation overhead and is intended for use when the processor configuration is not an issue. The warp PE model defines all instructions in the AIC instruction set as requiring 0 cycles to complete. Warp mode processing is a technique for simulating the presense of special-purpose hardware (such as support for trigonometric functions). The simulator separately accounts for the time to execute such functions.

The three remaining PE models are based on the processor reconfigurable architecture described in [6].

Program / Phase	Description
Filter	
Communicate 1	trade border values with neighbors
Median	median filter on image partition
Fix Corners	handle image borders
Communicate 2	trade border values with neighbors
Sobel	Sobel transform on image partition
Label	
Initialize	
Build Borders	determine pixels on region borders
Redistribute	load balance border pixels
Number Regions 1	distance doubling to find unique ID
Merge Borders	handle nested regions
Number Regions 2	distance doubling to find unique ID
Fill Regions	propagate region ID throughout region
Corners	
Unpack	data decoding
K-Curvature	check region border curvature at distance K
Smoothing	smooth curvatures with gaussian
Zero Crossings	find zero crossings in curvature
Mark Corners	corners have high curvature and zero crossing
Rectangles	
Initialize	
Eliminate Nodes	eliminate non-corner border pixels
Scan For Corners 1	eliminate regions with ¡ 3 corners
Convex Hull	eliminate corners not on the hull
Scan For Corners 2	eliminate regions with ¡ 3 corners
Find Right Angles	measure corner angles
Find Rectangles	rectangles have 3 consecutive right angles
Build Rectangle DB	extract rectangle parameters

Table 2: Programs and phases used in the experiments.

These models are identified simply as PE models 1, 2, and 3. *PE model 1* is a bit-serial processing element whose design is taken from the Connection Machine 1 [3], and is also used in the Connection Machine 2 and Connection Machine 200 (see Figure 5). This PE uses 1-bit data paths, has a small collection of 1-bit registers and a pair

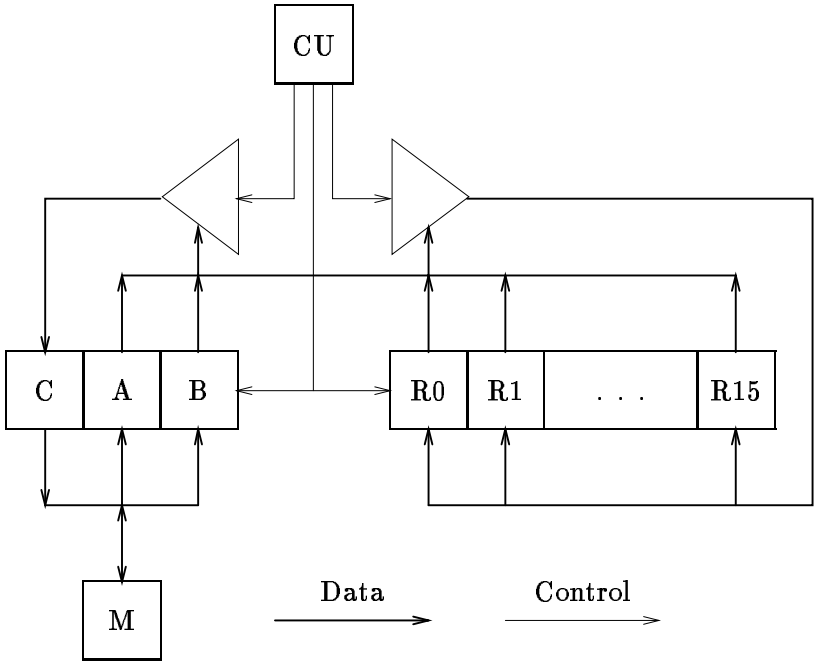

Figure 5: Architecture for PE model 1.

of identical functional units each capable of computing an arbitrary boolean function on three variables. The registers in the PE are partitioned into 3 I/O registers (A, B, and C) and 16 general-purpose registers. Two I/O registers A and B and one general purpose register provide the inputs to both the functional units. One output is stored in I/O register C, and the other in one of the general purpose registers. All transfers from memory are stored in register A or B, and all transfers to memory are from register C. The PE can perform both a memory transfer and a computation in a single cycle.

In Hillis [3], a prototype chip in 2 micron CMOS packaged 16 PEs of very similar design onto a single die with a network router. We have performed a logic-level design of the PE model 1 architecture and have found that 32 PEs and some reconfiguration hardware (described later) can easily be placed on a single 2 micron CMOS die 1.5 centimeters square. Simulation studies have shown that our design is capable of operating with a 20 nanosecond clock period. Given this level of performance, it is assumed that other system components such as memory and control signals present the limiting factor.

Computations using PE model 1 are performed bit-serially. Binary operations fetch one bit of each operand, compute one bit of result, and store that bit, saving one bit of internal state (such as carries) into a general purpose register. These operations can be performed in $1+3b$ cycles where b is the number of bits in the operands. More complex operations such as multiplication and floating point arithmetic are built up from integer and logical operations. For example, one floating point addition operation requires ≈ 4900 cycles.

PE model 2 is a simple bit-parallel architecture that can be constructed by using thirty-two model 1 PEs as a single unit. Thus a model 2 PE has 32-bit registers, and a pair of 32-bit boolean functional units. In addition, carry lookahead and barrel shifting logic is provided to perform these "global" functions. In comparison with the model 1 PE, a model 2 PE can perform 32-bit integer addition or subtraction using 5 cycles: two operand loads, compute carries, compute sums, store result. Complex instructions still require a number of simple instructions to be computed, but now all of these instructions can operate bit-parallel.

PE model 3 uses two model 2 PEs to allow multiple operations to be performed simultaneously in the course of a more complex instruction. This capability, together with the additional registers available, reduces the number of simple instructions that must be executed not only by performing parallel computations, but by allowing operands to remain in local registers, rather than requiring them to be stored to memory while a different part of the computation is performed. To perform this type of computation, the control system must be able to provide different control signals to cooperating components of the PE, and there must be high-speed data transfer capability between these components. Integer addition and logical operations perform no faster on model 3 PEs than model 2 PEs, but multiplication and floating point arithmetic do. As an example, model 3 PEs require ≈ 280 cycles to perform floating point addition, model 2 PEs require ≈ 370 cycles (see [6] for details).

PE models 1, 2, and 3 are specifically designed to be reconfigured from one into another. The reconfigurable systems we simulate utilize a constant amount of hardware, and simply reconfigure the logical view of the PEs.

For the experiments presented in this section, all of the architectures studied are SIMD machines. In all cases, it is assumed that exactly one instruction stream executes on a given CU, so context switching, scheduling, etc. are not an issue. The experiments described in this paper utilize an ICN model for a "k-ary n-cube" class of networks configured as a binary hypercube with serial links. The model is presented in [8] and is based on those used in Scott [9], Agarwal [1] and Dally [2]. The specifics of this model are beyond the scope of this paper.

4.2 Experimental Method

In [7] we outline an experimental methodology for utilizing RAW in the design of parallel architectures. In these experiments we utilize the same methodology. Briefly, we first study the performance of each of the four programs on an increasing number of *unit* model PEs, noting any change in the performance characteristic. Based on the results of this initial experiment, we select three distinct configurations of the processing elements (the details of which are given above in 4.1) and simulate the program in each configuration. Next we study the effects of processor reconfiguration by running each program using the configuration that provides the best performance. Finally, we repeat the last experiment, this time considering the various phases of each program and running each phase in the configuration that provides the best performance. In each case we include the costs of coordinating the various configurations. In these experiments, the costs are entirely due to architectural reconfiguration.

4.3 Experimental Results

Figures 6 A and B show the total execution time and execution profile of the filter program when executed on 1 to 16K unit model PEs. The first graph indicates clearly that the program is characterized by a large amount of usable parallelism. The profile shows the parallelism to be primarily due to the median and Sobel routines; the communication routines exhibit a lesser amount of parallelism. Figures 6 C and D indicate that PE model 1 performs the best for the median and Sobel routines by a factor of nearly 3, and all configurations are roughly equal for the communication routines. These results are to be expected. As stated previously, this is a fine example of a low-level vision routine and is exemplary of the fine-grain nature of that level of abstraction. One might expect the communication routines to be more significant for configuration 1 because as the number of pixels per PE is reduced, the ratio of communication to computation increases.

Figures 7 A and B show the total execution time and profile for the label program for 1 to 16K unit model PEs. These graphs indicate that the label program does not behave as well as the filter program. Note that while the filter program achieved a speedup of nearly four orders of magnitude for 16K PEs, the label program achieves less than 3 orders of magnitude speedup. Note also, that the graph for the label program curves up sharply between 4K and 16K PEs. An examination of the profile shows that this is primarily due to the number region step (which happens to be executed twice). The rest of the routines are fairly well-behaved, showing a speedup similar to that of the overall curve, except the initialization step, which shows over four orders of magnitude speedup as one might expect. Also note that a few phases (most obviously the merge phase) show a step-like behavior. This effect is due to the use of an algorithm that scans the image horizontally. As the

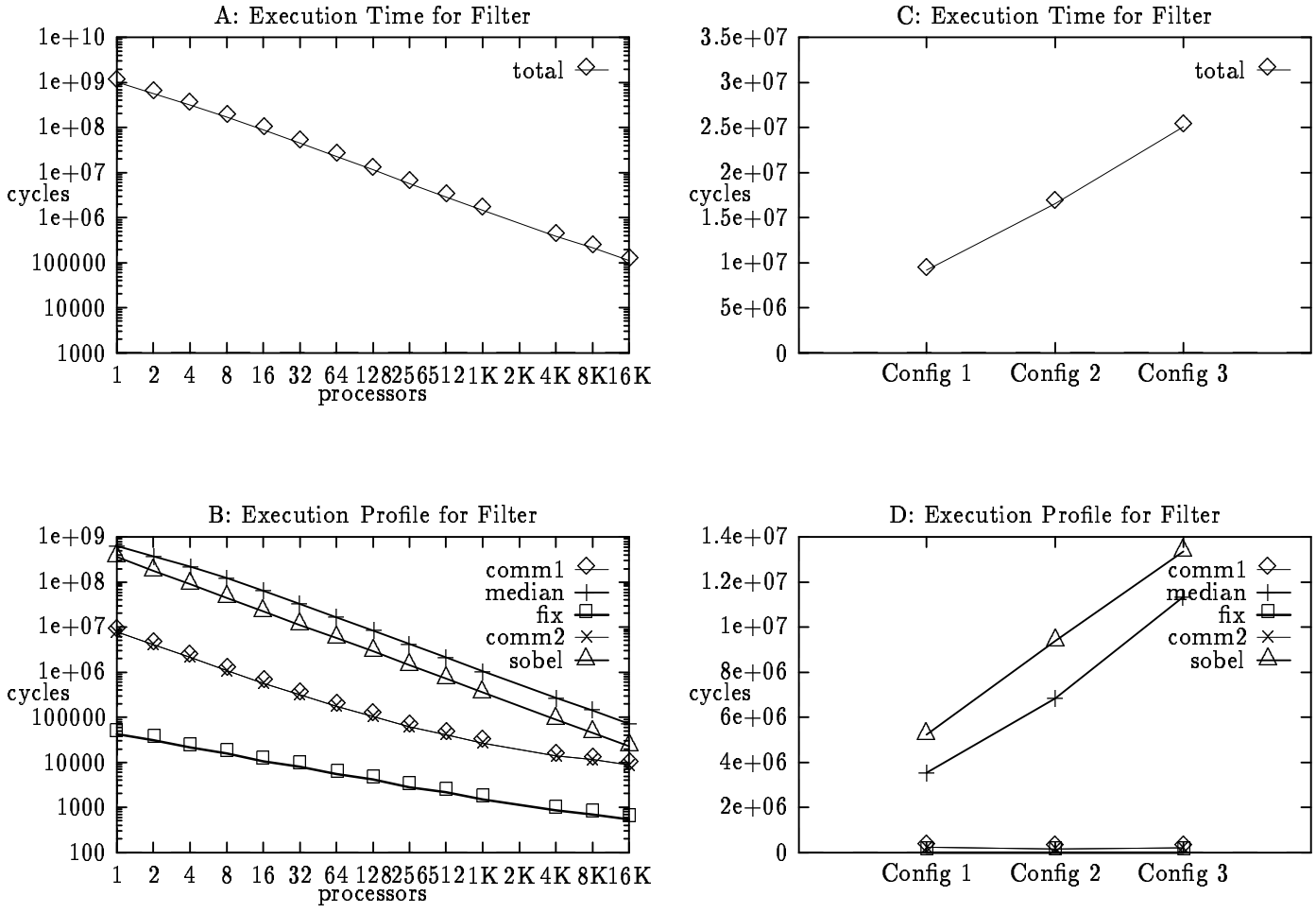

Figure 6: Execution time and profile for the filter program.

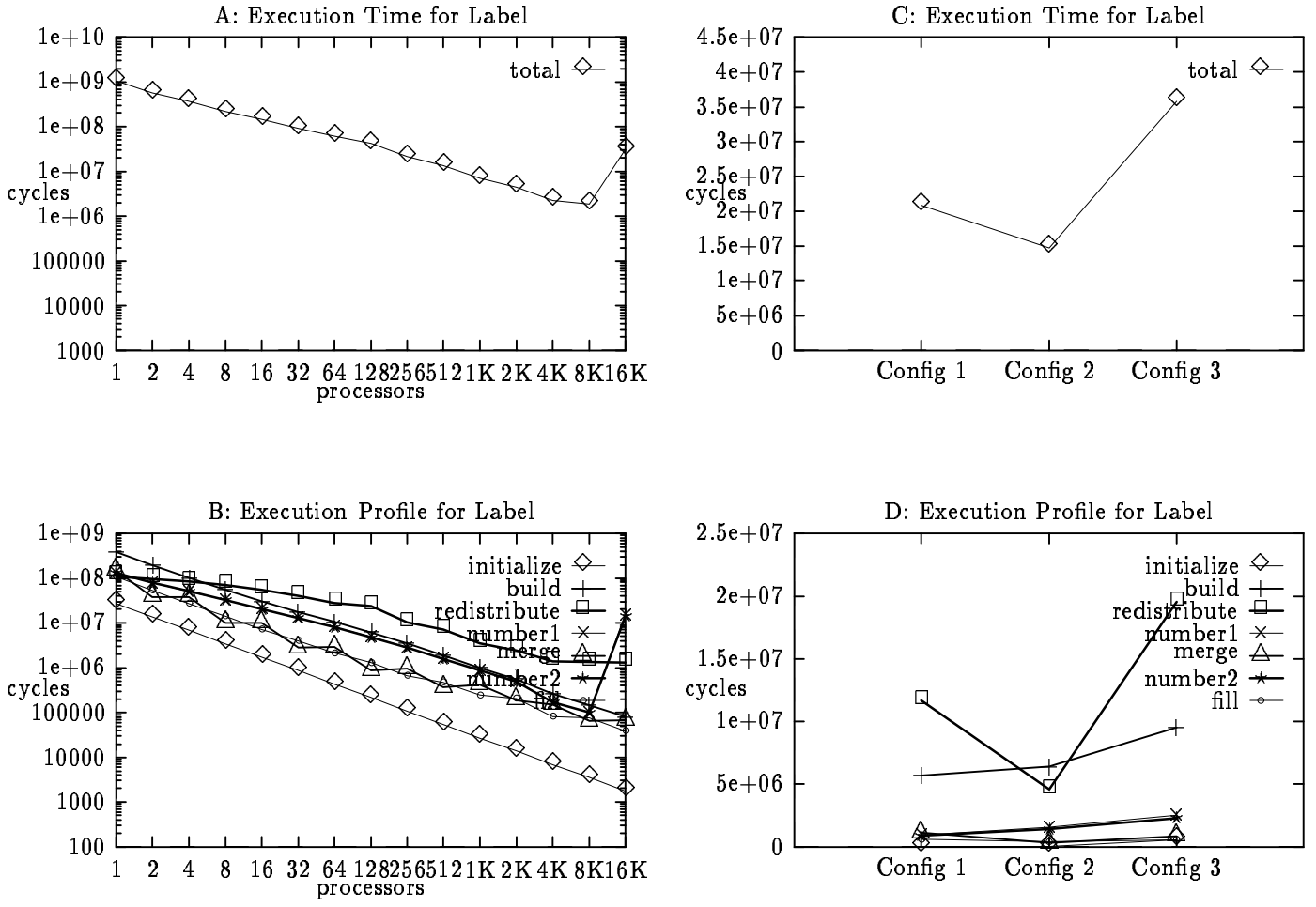

Figure 7: Execution time and profile for the label program.

number of PEs is increased, the number of pixels in the image partition for each PE is reduced first in the X dimension, then in the Y dimension. The horizontal scans work in the X dimension, so these transitions provide significantly more speedup. Also note that communication patterns in this routine are mostly random, therefore one cannot expect curves as smooth as that for the filter program.

Figures 7 C and D show the execution time and profile for the label program on the three static configurations. Here we see that configuration 2 performs the best for this program. This result occurs because the redistribute phase dominates the execution of this program. This phase is potentially erratic because its behavior depends on the distribution of border pixels among the PEs and it is communication bound. For a large number of PEs, the potential for a large deviation in the number of border pixels per PE can cause an increase in the amount of communication necessary. The other phases are split between those that perform best in configuration 2 and those that perform best in configuration 1. The build phase is similar to the filter program in its structure, and thus exhibits a similar characteristic. Oddly enough, the number regions phase also favors configuration 1. This program could benefit from processor reconfiguration, but as seen in the graphs, the benefits would not be dramatic, if noticeable at all.

Figures 8 A and B show the execution time and profile for the corners program for 1 to 16K unit model PEs. These results are a little surprising because the corners program has an order of magnitude less data than the filter program, and potentially erratic communication patterns. These results do make sense, though, when one considers that this program still maintains at least 2 data points per PE in the 16K PE case, and these data points have been conveniently distributed by the label program. As for the communication, this program requires much less than the label program because each point on a region border need only communicate with those pixels a small distance along the border, and in many cases these pixels are on the same PE, or a nearby one. Figures 8 C and D show the execution time and profile for the corners program on the three static configurations. Again, much like the filter program, this program favors configuration 1.

Finally, Figures 9 A and B show the execution time and profile for the rectangles program for 1 to 16K unit model PEs. Again, these results are rather surprising, as this program uses only about 350 data points! Like the label program, this program achieves much less speedup than the fine grain filter or corners programs: only about two orders of magnitude for 16K PEs. We also note that the curves are a little erratic, and there is a definite reduction in slope after about 128 PEs – probably due to the small number of data points. Oddly enough, though, this program does continue to improve its performance all the way up to 16K PEs. The explanation is that the data for this program is extremely unbalanced. It turns out the corners found in an image are necessarily very close in the image, thus the probability is high that many corners fall on a single PE, while

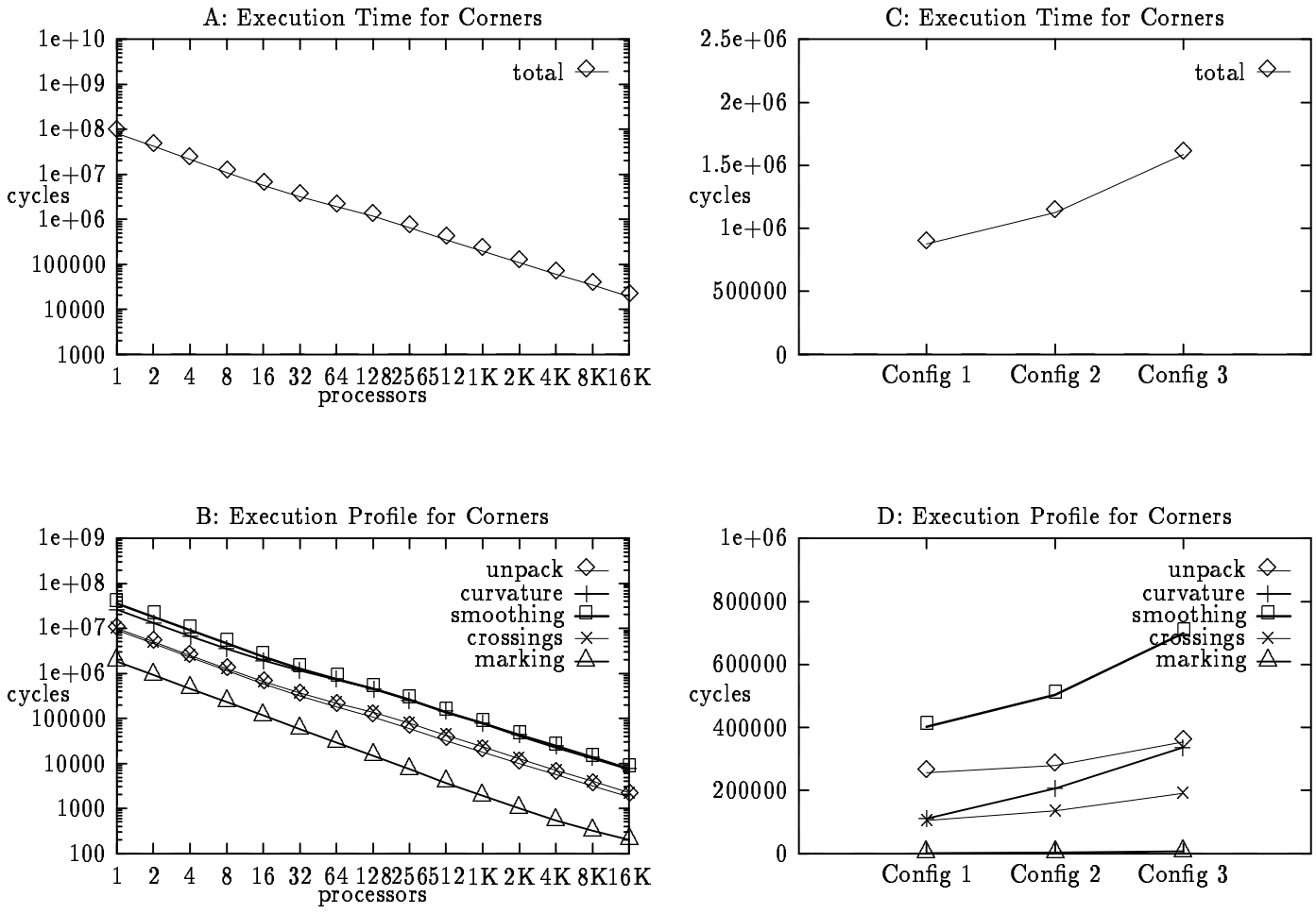

Figure 8: Execution time and profile for the corners program.

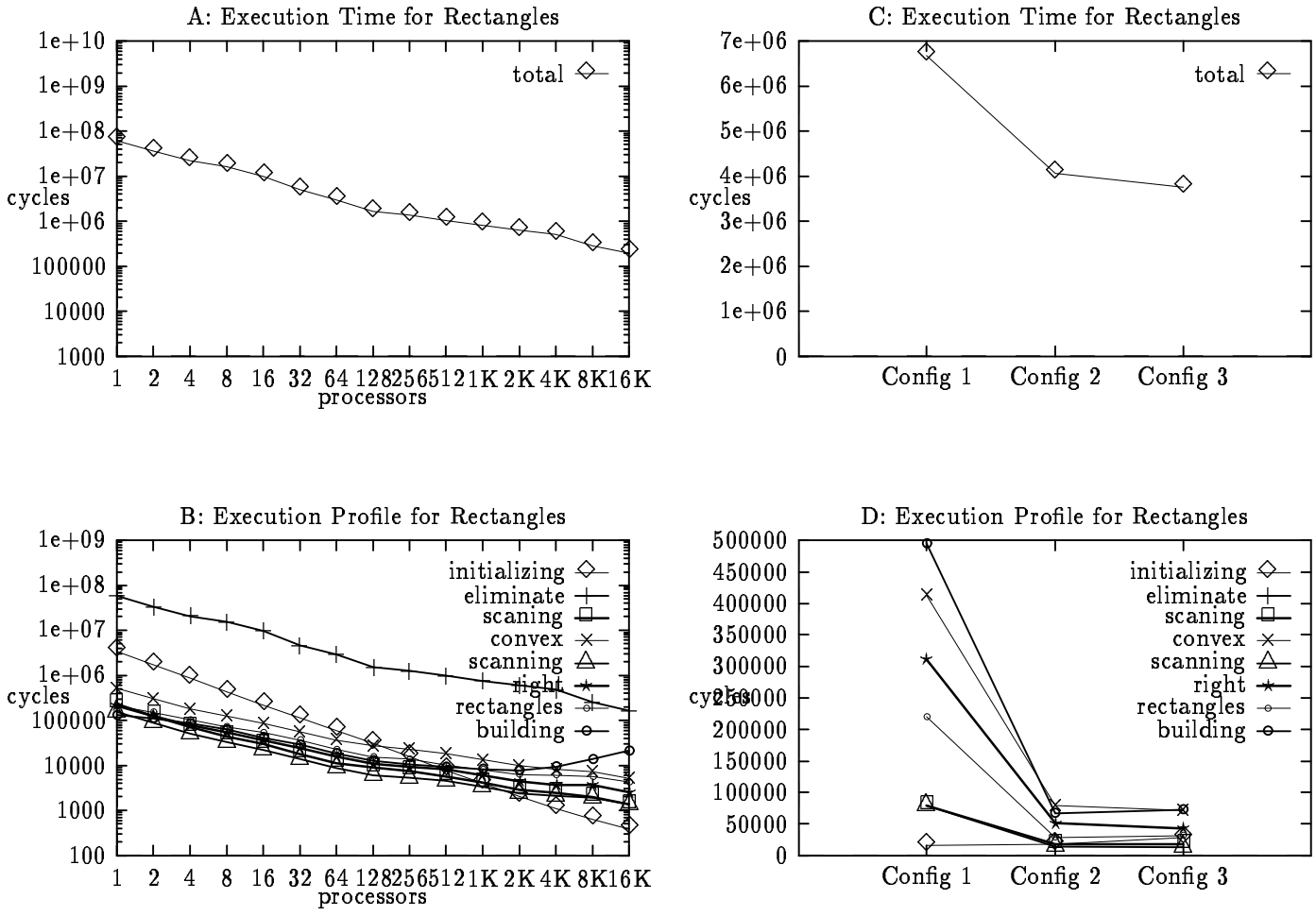

Figure 9: Execution time and profile for the rectangles program.

other PEs have no corner points at all. Thus, long before the number of PEs exceeds the number data points, many PEs are idle during most of the computation. As the number of PEs increases, the number of data points per PE decreases, but at a considerably slower than the number of PEs in increasing. Each time more PEs are added, most of the new PEs are idle. This explains the extremely shallow slope of the graph. We note that a redistribution algorithm would likely alter these curves dramatically, but we add that the dominant phase of the program, the eliminate phase, is the one charged with eliminating the non-corner nodes from the boundary lists at the beginning of the program and thus uses as many data points as the corners program.

Figures 9 C and D show the execution time and profile for the rectangles program on the three static configurations. Here we see an expected result, that the rectangles program indeed favors configuration 3 by a wide margin over configuration 1. An examination of the profile shows this to be true for all phases save the initialization phase, though some of the other phases perform as well or even slightly better using configuration 2. This result is reasonable because clearly the program makes only modest use of additional processing elements, but the complexity of its computations can easily make effective use of more powerful processing elements. As previously stated, this is a classic example of middle level vision.

Now we consider the execution of these four programs together as a system for identifying rectangles in an input image. First we show the execution time of the system for each of our three configurations in Figure 10. This figure shows that the effect of the filter and label program completely overshadow that of the rectangles program, and further that the label program's characteristic dominates that of the filter and corners program, to indicate that configuration 2 is the fastest of the configurations. Clearly, there is enough variation in the system that processor reconfiguration is worthy of consideration. Figure 10 also shows the performance of a processor reconfigurable architecture on the rectangle extraction system. There are two points plotted. The first shows the execution time for the rectangle extraction system allowing reconfiguration between the four different programs as shown in Table 3. This results in about a 20% improvement in performance. The second point shows the

Program	Configuration
Filter	1
Label	2
Corners	1
Rectangles	3

Table 3: Configuration used by each program in IU benchmark.

execution time of the rectangle extraction system allowing reconfiguration between the phases of the programs as shown in Table 4. Here, a nearly 30% improvement is achieved.

Program / Phase	Configuration
Filter	
Communicate 1	1
Median	1
Fix Corners	1
Communicate 2	1
Sobel	1
Label	
Initialize	1
Build Borders	1
Redistribute	2
Number Regions 1	1
Merge Borders	2
Number Regions 2	1
Fill Regions	1
Corners	
Unpack	1
K-Curvature	1
Smoothing	1
Zero Crossings	1
Mark Corners	1
Rectangles	
Initialize	1
Eliminate Nodes	1
Scan For Corners 1	3
Convex Hull	3
Scan For Corners 2	3
Find Right Angles	3
Find Rectangles	2
Build Rectangle DB	2

Table 4: Configuration used by each phase of IU benchmark programs.

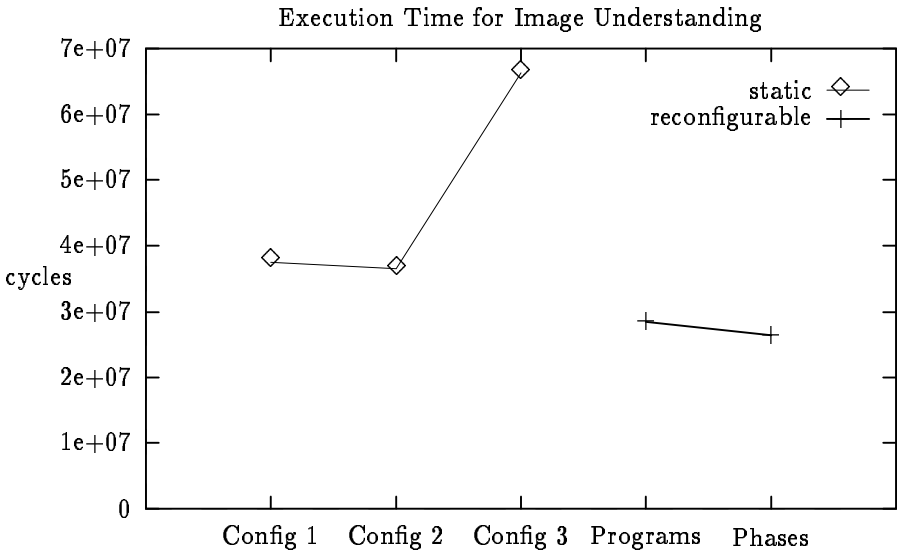

Figure 10: Execution time on three static configurations for all four routines.

4.4 Discussion

The important aspect of this experimentation is to reveal interesting phases that any given applications (computer vision in this instance) may exhibit which may have direct implications on the choice of architecture for that phase. The results we present here demonstrate that multigauge heterogeneous parallel architectures can achieve performance gains over static parallel systems in low and middle level vision applications. There are several points the reader should note. First, these performance gains were achieved by exploiting differences in a basic characteristic of the programs studied, as is discussed at the beginning of this section. High level vision routines exhibit an even greater disparity in this same characteristic, and would thus seem to be reasonable candidates for future experimentation. Next, the architectures studied in these experiments are reconfigurable architectures, which implies that they benefit from a maximum degree of flexibility, but suffer in that they require expensive custom hardware to implement. In particular, the fact that the same computational resources are used in each architectural configuration means that interconnection issues are less significant in contributing to the cost of coordinating the various phases than would be the case if a non-reconfigurable heterogeneous system were employed. Thus, a second avenue of future research would be to explore the tradeoffs present between these different approaches. As was previously discussed, the PCI model is equipped to study such types of systems as well. Finally, ultimate goal of our research it to attain a higher level of understanding of

the program/architecture relationship through experience gained through such experimentation. These results provide one set of data points, we need many more covering a wide range of application domains and architectural features in order to build a clearer picture of this relationship that would lead to our desired understanding.

5 Conclusion

A clear understanding of the performance relationship of parallel programs and parallel architectures is essential to the successful implementation of heterogeneous parallel systems. We have taken one step in the development of this understanding by proposing the PCI model of parallel programs and parallel architectures which provides a basic means for reasoning about this relationship. Our initial work with this model has been to utilize it in experimenting with the performance implications of multigauge heterogeneous systems in the form of processor reconfigurable architectures. To do this, we have implemented RAW, a simulation environment based on the PCI model, which facilitates experimentation. Our experiments examine the use of processor reconfigurable architectures in low and mid-level vision applications and show that such multigauge architectures provide better performance than any one homogeneous static architecture. Our ongoing research addresses other aspects of heterogeneous system design in thge context of computer vision such as interconnections (both multiprocessor and networked multicomputers), different control regimes (such as SIMD, MIMD, and MSIMD), and multigranular systems composed of networked parallel systems of different granularities.

References

[1] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] W. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[3] W. D. Hillis. *The Connection Machine.* The MIT Press, Cambridge, 1985.

[4] S. I. Kartashev and S. P. Kartashev. A multicomputer system with dynamic architecture. *IEEE Transactions on Computers*, C-28(10):704–721, October 1979.

[5] W. B. Ligon. *An Empirical Analysis of Reconfigurable Architectures.* Doctoral Thesis, Georgia Institute of Technology, Atlanta, 1992.

[6] W. B. Ligon and U. Ramachandran. A Reconfigurable Supercomputer Architecture. Technical Report GIT-ICS-89/13, Georgia institute of Technology, February 1989.

[7] W. B. Ligon and U. Ramachandran. An Empirical Methodology for Exploring Reconfigurable Architectures. *to appear in Journal of Parallel and Distributed Computing*, 1992.

[8] W. B. Ligon and U. Ramachandran. Simulating Interconnection Networks in RAW. Technical Report ECE-102692-0915P, Clemson University, September 1992.

[9] S. Scott and J. Goodman. Performance of pipelined-channel k-ary n-cube networks. Technical Report CS-1010, University of Wisconsin - Madison, 1991.

[10] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski. An overview of the Texas Reconfigurable Array Computer. In *Proc. 1980 AFIPS Nat. Comput. Conf.*, pages 631–641, Arlington, May 1980. AFIPS Press.

[11] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley Jr., and S. D. Smith. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C-30(12):934–947, December 1981.

[12] L. Snyder. Introduction to the Configurable, Highly Parallel Computer. *IEEE Computer*, pages 47–56, January 1982.

[13] C. Weems, A. Hanson, E. Riseman, and A. Rosenfeld. An integrated image understanding benchmark. Technical report, University of Massachusetts, in preparation.