FILE COPY

AD-A232 114

REPORT DOCUMENTATION PAGE

	REPORT DATE DECEMBER 1990	REPORT TYPE AND DATES COVERED ~~THESIS~~/DISSERTATION

TITLE AND SUBTITLE

Data Type Coherency in Heterogeneous Shared Memory Multiprocessors

AUTHOR(S)

Michael W. Strevell

PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

AFIT Student Attending: Univ of Texas - Austin

AFIT/CI/CIA-90-D032

SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFIT/CI
Wright-Patterson AFB OH 45433-6583

SUPPLEMENTARY NOTES

DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release IAW 190-1
Distributed Unlimited
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer

NUMBER OF PAGES
140

DATA TYPE COHERENCY IN HETEROGENEOUS SHARED MEMORY MULTIPROCESSORS

by

MICHAEL W. STREVELL, B.S.E.E., M.S.E.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1990

DATA TYPE COHERENCY IN HETEROGENEOUS SHARED MEMORY MULTIPROCESSORS

APPROVED BY DISSERTATION COMMITTEE:

[illegible]
Harvey G. Cragon, Supervisor

[illegible]
James C. Browne

[illegible]
Mario J. Gonzalez

[illegible]
G. Jack Lipovski

B.F. Womack
Baxter F. Womack

91 2 06 087

Dedication

I dedicate this work to

my Lord and Savior, Jesus Christ.

I thank him for his work in presenting me blameless before God,

and pray that all my work would glorify him before others.

Acknowledgements

I am deeply grateful to Harvey Cragon for his guidance and support during my graduate career here at The University of Texas. His positive approach and Theory Y leadership style made my studies here a true pleasure.

I thank J.C. Browne, Mario J. Gonzalez, G. Jack Lipovski, and Baxter F. Womack for their interest in my research, and for their suggestions which have benefitted this research.

I thank Julie Hoe for her careful reading of this dissertation and her suggestions which improved the clarity and readability.

The United States Air Force Academy supported me for my Masters and PhD. I would like to particularly thank Brigadier General Erlind Royer, the Dean, and Colonel Alan Klayton, the Head of the Department of Electrical Engineering. Their assistance in working out my assignments made this possible.

DATA TYPE COHERENCY IN HETEROGENEOUS SHARED MEMORY MULTIPROCESSORS

Publication No.____________

MICHAEL W. STREVELL, Ph.D.

The University of Texas at Austin, 1990

Supervising Professor: Harvey G. Cragon

A heterogeneous shared memory multiprocessor, which contains different types of specialized processors, may execute a complex problem faster than either a homogeneous multiprocessor or a heterogeneous network. However, since dissimilar processors often use different representations for primitive data types, the shared data must be transformed. Analytical performance models and queueing models predict the performance of alternative designs. These models indicate that significant performance advantages are provided by hardware transformation units, caching of unshared data, and local memory. Conversely, caching of shared data and the location of the transformation units have a less significant effect on performance. The primary applications for these type of designs are in special purpose applications which require maximum performance and tight coupling between heterogeneous processors. The linking that must be done at compile time makes these designs less suited for general purpose applications and development work.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Virtually all parallel processors that have been built or proposed to date use only a single processor type. However, the ability to interconnect different types of specialized high performance processors would allow the design of custom parallel architectures for specific purposes. Heterogeneous parallel processors that are specially suited to specific problems could offer a significant performance advantage. For maximum performance, the ability tc share memory is desirable. But since different processors use different methods of data representation for the primitive data types, the values in memory cannot be shared directly. The shared data must be transformed into the appropriate representation for each processor.

In addition to the potential performance advantage, a heterogeneous system provides greater versatility. In situations where a complex application has been designed for a specific processor, it may be preferable to add that processor to the system rather than port the application to a new processor.

Most of the prior research has focused on versatility and has addressed the communication between heterogeneous computer systems at the network level. Some of these projects include Mach [Rash87] [Rash88], XDR [XDR86], Matchmaker, Agora [Bisi87], Nectar [Arno89], and Cronus [Dean87] [Scha88].

In a project conceptually similar to this research, MITRE developed VLSI coprocessors to accelerate translations between network protocols [LAN 87]. One of the significant performance problems encountered in this project resulted from the overhead required to set up the transformation hardware [Frie88]. Their observations on the importance of minimizing overhead strongly influenced the early development of this research.

This research focuses on heterogeneous shared memory multiprocessors designed for maximum performance. Performance is emphasized for two reasons. First, most of the previous work has addressed versatility rather than performance. The purpose of this research is to help fill this void by proposing the use of hardware transformation units that will enable shared memory access in heterogeneous multiprocessors [StCr88]. Shared memory architectures were chosen because this is the fastest way to share data between processors. These architectures are well-suited for special purpose applications that require high speed execution, such as embedded computer systems. An embedded computer system is one in which the computer is designed into an end item such as an aircraft, a missile, communications or test equipment, etc. Because of the emphasis on embedded systems, this research has tended to explore those design alternatives that favor reduced run time hardware and processor cycles and instead shift the burden to development and compile time. This research does not specifically address general purpose computer systems which often have greater emphasis on versatility than performance.

The second reason for emphasizing performance is because this author has observed unique military applications that would benefit from tightly integrated, high performance, heterogeneous multiprocessors. Some of these projects include the Advanced Tactical Fighter (ATF), Joint Integrated Avionics Working Group (JIAWG), Pilot's Associate, and autonomous vehicles. The Department of Defense and the defense industry [Heil87] are aware of the importance of this problem. The Air Force, Army, and Navy are all developing advanced aircraft that will have much tighter integration between dissimilar processors [JIAW88] [JIAW89], although the details of the competing designs are still heavily shrouded in proprietary security [ATF89].

In this research, a heterogeneous shared memory multiprocessor has two characteristics. First, it contains different types of autonomous processors. Under Flynn's taxonomy, this would be classified as a Multiple Instruction Multiple Data (MIMD) architecture. Each processor must be capable of fetching and executing an independent instruction stream. Secondly, the processors must be able to access shared memory. This characteristic differentiates these architectures from a distributed computer system or a network of computers.

Analytical performance models are used to compare alternative heterogeneous designs. The performance metric for this comparison is the average memory access time. The intent of this research is not to espouse a certain design, but rather to provide predictive models of performance which can be used by a designer of heterogeneous architectures.

The only current heterogeneous shared memory multiprocessor known to the author is the Explorer LX by Texas Instruments [ExLX87]. This system has two processors, an Explorer LISP processor and a Motorola 68020, that share memory. The XDR protocol (§3.3.1.) is used to transform the data.

However, the concept of specialized processors that share main memory is not new to the history of computing. Computer architects have used heterogeneous shared memory multiprocessors since the mid-sixties to take advantage of functional specialization and high bandwidth communication. The IBM System/360 [Hell67] and the Texas Instruments Advanced Scientific Computer (ASC) [CrWa89] both used specialized stored-program processors to control the movement of data between peripheral devices and main memory. These peripheral processors used the same primitive data types as the main processor, which reduced the need for data type transformations.

However, specialized processors can perform functions other than data movement and simple control. An example of this is described in the next section. But this heterogeneity brings with it the problem of data type portability between processors. If the processors use dissimilar representations, the data types must be transformed. This research proposes the use of a common data representation for shared memory, and hardware transformation units for each processor in order to provide low latency and high bandwidth communication to memory.

1.1. Advantage of Heterogeneous Multiprocessors

Some complex problems include a broad spectrum of tasks. An advanced fighter aircraft requires a mixture of processors for digital signal processing (DSP), graphics, and general purpose (GP) data (Figure 1-1). In situations such as this, where high performance is vital, specialized processors provide a significant performance advantage over general purpose processors. Specialized processors exploit the structure of a task and its data, and can execute that task faster and often cheaper than a general purpose processor. Lipovski and Malek [LiMa87] make a similar argument from the standpoint of power and energy:

> A well-designed special-purpose computer should require much less power since only the hardware needed by the procedure need be built. It should execute faster because no run-time scheduling is necessary and direct paths for data transfer should be implementable.

Specialized processors have been developed for digital signals, graphics and images, floating point numbers, list manipulation, data bases, radar, electronic countermeasures, general purpose data, and others. Digital signal processors have historically been eight to ten times faster at DSP than general purpose processors [Morr86] [Morr88]. Similarly, the TMS34010 pixel addressable graphics processor is estimated to be 16 to 20 times faster at graphics than current general purpose processors [Chan90] (Figure 1-2). Ideally, a heterogeneous multiprocessor with the performance advantages shown in Figure 1-2 can execute the workload in Figure 1-1 in approximately one third the time required by a general purpose processor (Figure 1-3). The execution times shown in Figure 1-3 are computed from the workload in Figure 1-1 and the performance comparison shown in Figure 1-2. This relative execution time is based on the equation:

$$\text{Execution Time} = \sum_{i=0}^{2} \frac{\text{Workload fraction}_i}{\text{Speedup}_i}$$

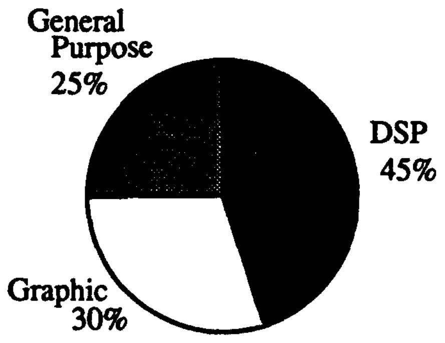


Figure 1-1: Advanced aircraft workload breakdown

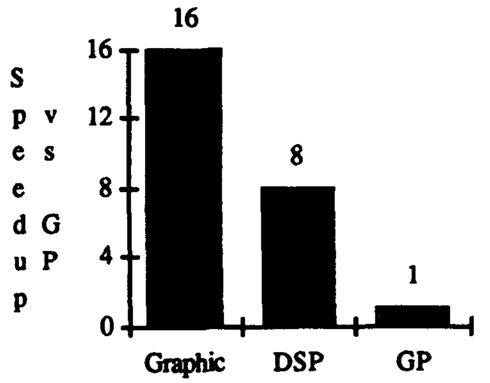


Figure 1-2: Performance advantage of specialized processors relative to general purpose (GP)

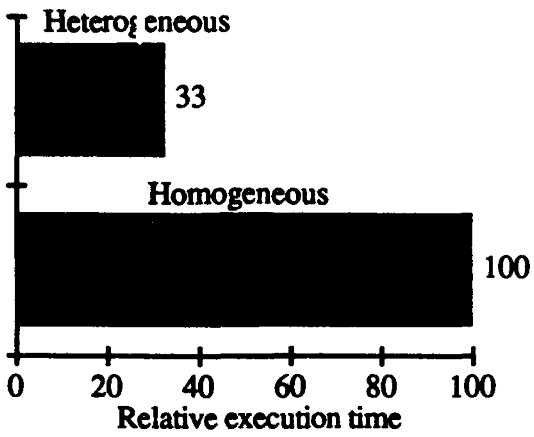


Figure 1-3: Execution time comparison of heterogeneous and homogeneous multiprocessors for the performance data of Figure 1-2

In practice, the overhead involved in sharing data between heterogeneous processors will somewhat reduce the actual performance advantage. Depending on whether message passing or shared memory is used, this overhead may consist of processor delay, data latency, memory contention, transformation time, and cache invalidations and/or updates. If the frequency of shared memory access is low, message passing and software transformations may be more cost effective. Shared memory and hardware transformation units will be advantageous only when the frequency and speed of shared memory accesses offset the cost of the shared memory and transformation hardware. The performance models in this research provide the computer designer information on the performance of shared memory and hardware transformation units.

Designers are beginning to recognize the performance advantages of heterogeneous multiprocessors. At a recent symposium, Motorola presented a design that has an MC88100 RISC processor pipelined with two DSP96002 DSP processors for a graphics application which uses floating point numbers [Serr90]. In this case, transformations were not required since both types of processors use the same IEEE floating point representation.

1.2. Dissertation Organization

Chapter 2 describes the need to maintain coherency for data values, addresses, and data types. Cache coherency is more complex in a heterogeneous multiprocessor because the representation of the data may vary between caches. Coherency can be maintained through the use of transformation units, identical cache line sizes, compatible cache coherency protocols [SwSm86], and compatible addressing schemes. The problems involved in designing multiprocessors containing both early- and late-binding processors are also discussed.

Chapter 3 examines the underlying problems related to the representation and transformation of primitive data types. The types of transformations are categorized, and a common data representation is proposed. The transformation errors that occur between floating point representations are explained.

Chapter 4 examines alternative locations and methods for performing data type transformations. The performance of shared memory and hardware TUs is compared to networks and transformations in software.

Chapter 5 uses analytical performance models and queueing models to compare the performance of alternative designs using shared memory and hardware transformation units. These models identify the design features that provide the greatest performance improvement.

Chapter 6 briefly discusses the system software required to support these heterogeneous architectures. A heterogeneous linker creates a master symbol table containing the addresses and data types of shared data. Post-processors use the information from this table to insert transformation unit setup instructions into the compiled code for each processor.

Finally, Chapter 7 summarizes the conclusions of this research and its contributions to the design of high performance heterogeneous multiprocessors.

Chapter 2

Data Coherency

A heterogeneous multiprocessor must maintain coherency for data values, addresses, and data types. Data value coherency requires compatible cache coherency protocols. Address coherency is maintained through the use of fully aligned memory storage, with specialized hardware for processors with smaller address and data buses. Data type coherency is accomplished through transformations, which are covered in the next chapter. When a multiprocessor includes both early and late binding processors, some of the data type transformations are not known at compile time and must be determined at run time.

2.1. Value Coherency

Heterogeneity imposes additional requirements on the cache (value) coherency scheme. Many cache coherency protocols have been proposed and compared in the literature [ArBa86] [BiDe86] [MiBa89]. All of these protocols are for homogeneous multiprocessors, and they often rely on monitoring a common bus. Sweazy and Smith [SwSm86] define a class of compatible consistency protocols supported by the IEEE Futurebus. Three of their conclusions which are relevant to heterogeneous multiprocessors are:

> (1) [They] define a class of compatible protocols, such that each cache in the system may implement one of the protocols in this class and still maintain consistency with other caches implementing different (compatible) protocols. This permits the coexistence of copy back caches, write-through caches and non-caching boards in the same system [SwSm86].
>
> (2) All caches must use the same line size.

(3) To implement the protocols, six signal lines are required on the bus. Certain other protocols require a seventh signal.

A heterogeneous system is unique in that the cache representation (which is determined by the location of the transformation unit) affects the performance and thus, the choice of the coherence protocol. A protocol which *invalidates* shared lines in other caches is not affected by the cache representation. However, a protocol which *updates* shared lines in other caches will be delayed by a native representation cache. A native representation cache requires a transformation when a shared line is updated, while a common representation cache does not require a transformation for updates (§5.3).

2.2. Address Coherency

Both the address and data lines are involved in maintaining coherency. Since all protocols maintain coherency using the address of the cache line, all processors which access shared memory must use the same addressing scheme (coherent addresses).

The address coherency problem arises due to the fact that processors store bytes in memory in one of two different orders. This is commonly referred to as the Big-Endian (BE) vs. Little-Endian (LE) problem from a well-known article by Cohen [Cohe81]. This problem is further analyzed in [Kirr83]. The BE processor stores the *most* significant byte of the word at the lowest address, while the LE processor stores the *least* significant byte of the word at the lowest address (Figure 2-1 and 2-2) [MIPS88]. Address coherency problems between BE and LE processors (§3.1.1.) are caused by data that is not aligned on 32-bit word boundaries. When addressing a misaligned word with a byte address of 3, Figure 2-3 shows the bytes that must be accessed for each of the two conventions. These problems can be eliminated by requiring full alignment of all data in shared memory (as discussed in §3.3.4.). When accessing an aligned 32-bit word in memory, both BE and LE processors put out the same address – the byte with the lowest address. However, the BE and LE processor are referring to different bytes in the word. Nevertheless, an aligned address will return the same four bytes with either processor, although the four bytes in the word will be in the opposite order. The byte order can be corrected by a transformation unit.

Big-Endian

31 24	23 16	15 8	7 0	Word address
8	9	10	11	8
4	5	6	7	4
0	1	2	3	0

Most significant byte (byte 3) is at lowest address
Word is addressed by byte address of most significant byte

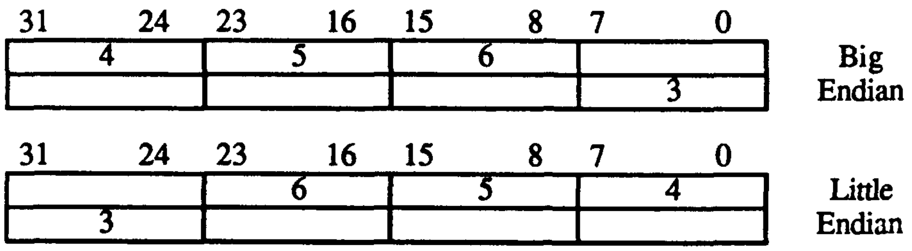
Figure 2-1: Addresses of bytes within words for Big-Endian

Little Endian

31 24	23 16	15 8	7 0	Word address
11	10	9	8	8
7	6	5	4	4
3	2	1	0	0

Least significant byte (byte 0) is at lowest address
Word is addressed by byte address of least significant byte

Figure 2-2: Addresses of bytes within words for Little-Endian



Bytes accessed when addressing a misaligned word with a byte address of 3

Figure 2-3: Bytes accessed for a misaligned word

A second type of address coherency problem occurs between processors with different address lengths and "addressable data unit" lengths [BlBr87]. For example, the MIL-STD-1750A has a 16-bit address and a 16-bit addressable data unit (i.e., word addressable, where a word=16 bits), whereas the R3000 has a 32-bit address and an 8-bit addressable data unit (byte addressable). So every address on the 1750A requires two addresses on the R3000 to hold the two bytes of data (Figure 2-4).

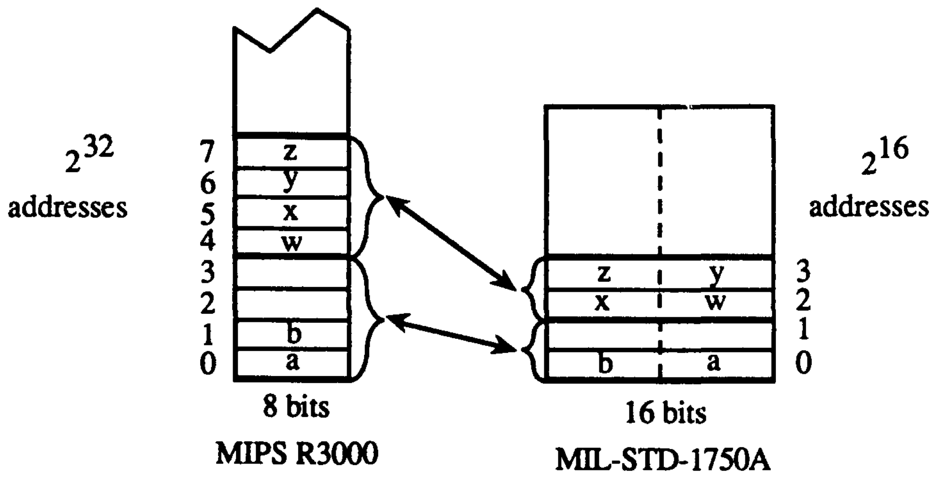


Figure 2-4: Address length vs. addressable data unit length

To share memory between these two processors, an address conversion must be performed. A portion of the address space for the 1750A can be mapped to the shared memory. When the 1750A puts out an address which is in that range assigned to shared memory, a decoder could cause 14 bits to be appended to the upper end of the 16-bit address, and 2 bits to the lower end, to create a 32-bit address. The 2 bits appended to the lower end are set to zero and ensure that the addresses generated by the 1750A are four bytes apart, which maintains memory alignment on word boundaries. The bits appended to the upper end control the location of shared memory. These bits can be appended using hardwiring or a buffer. A buffer allows the mapping to be software controlled.

2.3. Type Coherency

A third aspect of coherency is data type coherency. Data types and their representations vary greatly between processors. This problem and a solution are considered in detail in the next chapter. Here we consider the effect of binding time on type coherency.

2.3.1. Binding Time

When passing data from one routine (the source) to another routine on a dissimilar processor (the destination), two transformations are necessary: one from the source representation to the shared memory (SM) common representation, and a second from the shared memory common representation to the destination representation (Figure 2-5). Of course, if either the source or destination processor use the same representation as shared memory, that processor can access shared memory without a transformation.

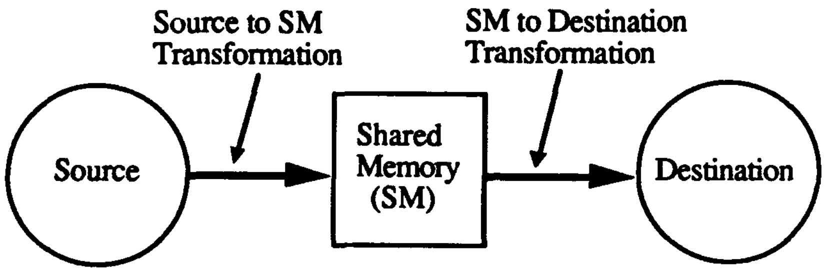


Figure 2-5: Source to destination transformations

In order to determine the appropriate transformation in an untagged environment, the data type of the shared data must be known at compile time so that the appropriate transformation instructions can be inserted in the code (§6.1). This is normally not a problem, since most languages assign, or "bind" the data type when the program is compiled. These are called early binding languages. However, some languages such as LISP are late binding, which means that the data type is not assigned until run time.

- **Early binding** is done at **compile time**
- **Late binding** is done at **run time**

A late binding time makes it impossible to completely specify both transformations at compile time. The simplest solution is to force late binding routines to do early binding for shared data; Common LISP permits this early binding. In order to allow late binding of shared data, the late binding routine must be capable of selecting the appropriate transformation at run time.

The four possible cases based on binding time and source or destination of the data are: early to early, early to late, late to early, and late to late (Table 2-1).

When sharing data between two early bound routines, the data types at the source and destination are known at compile time, so the heterogeneous linker (§6.1) can determine the necessary transformations.

Binding Time Combinations	
Source	Destination
Early	Early
Early	Late
Late	Early
Late	Late

Table 2-1: Binding time combinations

In the early to late case, the data type at the source is known but not at the destination. With a common representation shared memory, the source-to-shared memory transformation can be specified at link time. But the shared memory-to-destination transformation must be specified by the late-binding routine at run time. The heterogeneous linker passes the shared memory data type to the late binding destination routine at link time.

The late to early case is analogous to the early to late case: the data type at the destination is known but not at the source. Since the setup command for the early binding destination must be inserted at compile time, the shared memory data type must be specified at compile time. This in turn restricts the output of the transformation which the late binding source selects. The heterogeneous linker passes the shared memory data type to the late binding source at link time.

In the late to late case, the data type is not known at the source or the destination at compile time. Late binding languages normally use data type tags to pass data type information to other routines at run time. However, if the shared memory common representation does not have tags, the data type information must be communicated in some other manner. Either the shared memory data type must be specified by the linker at compile time, or the data must be passed through the shared memory unchanged so as to preserve the tags (§3.3.4. "opaque data").

Chapter 3

Data Type Representation and Transformations

Most processors have a similar set of primitive data types but they use different methods to represent them. The representation is important because the instruction set on a processor is designed to recognize and manipulate primitive data types represented in specific ways. Different representations make it impossible for heterogeneous processors to directly share data.

A *primitive data type* for a given processor is one which is represented in hardware and is manipulated by the instruction set. The three most prevalent primitive data types are signed integers, unsigned integers, and floating point numbers. Other data types, which are often used in software but are not usually primitive data types, are characters, vectors, arrays, strings, and complex numbers. Some scientific computers such as the Cray and the ASC have vector and string primitive data types.

Various numerical methods are used to represent primitive data types. These include two's complement, sign magnitude, exponent bias, hidden bit, the IEEE floating point standard, and others. The representation of floating point numbers has been quite diverse historically, as shown in Table 3-1. This diversity has made it difficult to share data between processors, or even get the same numeric result when a program is ported to another computer. But the recent development of the IEEE Floating Point Standard [IEEE85] now makes it easier for processors that use this standard to share floating point data.

Table 3-1: Floating point representation methods

Processor	Total Bits	Exp Bits	Mant Bits	Exp Rep	Mant Rep	Other
IEEE-S	32	8	24	+ 127	SM,NHB	
IEEE-D	64	11	53	+ 1023	SM,NHB	
VAX-F	32	8	24	+ 128	SM,NHB	OMZ=0
VAX-D	64	8	56	+ 128	SM,NHB	OMZ=0
VAX-G	64	11	53	+ 1024	SM,NHB	OMZ=0
VAX-H	128	15	113	+16384	SM,NHB	OMZ=0
CDC	60	11	49	+ 1024	OC,NI	
Harris	47	8	39	SM		
TMS320C30	32	8	24	TC	TC,NF	
TMS320C30	40	8	32	TC	TC,NF	
1750A-S	32	8	24	TC	TC,NF	
1750A-E	48	8	40	TC	TC,NF	
Explorer			Same	as IEEE		
MC68882			Same	as IEEE		
80387 S,D			Same	as IEEE		
80387 E	80	15	65	+16383	NF1	
AM29027		Supports IEEE; IBM S & D; and Vax D, F, & G				
IBM 360-S	32	7	25	+ 64	SM,NF	
IBM 360-D	64	7	57	+ 64	SM,NF	
Cray	64	15	49	+16384	SM	

SM	-	Sign Magnitude
HB	-	Hidden Bit
NHB	-	Normalized, $1.0 \leq x < 2.0$, Hidden Bit (the one is implied)
NF	-	Normalized Fraction, $0 \leq x < 1.0$
NF1	-	Normalized Fraction, $1.0 \leq x < 2.0$
OMZ	-	Order of Magnitude Zero
OC	-	One's Complement
NI	-	Normalized Integer (binary point at the right of bit 0)
TC	-	Two's Complement

Heterogeneity between processors can be accommodated in two ways: by defining a standard data representation for all future processors, or by using transformations between dissimilar representations.

A standard data representation may be the preferred long term choice because it avoids the problems and incompatibilities that arise in data type transformations.

The IEEE Floating Point Standard [IEEE85] is one such standard representation that is now being widely used in new processors; this standard will simplify the exchange of floating point numbers. Although in some cases a specialized application may require a unique representation, heterogeneity does not necessarily require a different data representation. For example, the TI TMS320C30 DSP processor uses a two's complement floating point representation for computing efficiency and reduced complexity, whereas the Motorola 96002 DSP processor uses the IEEE standard for compatibility with other processors.

In spite of the inherent simplicity of a standard data representation, no general data standard is currently in existence or even on the horizon. Thus it appears that transformations will be necessary for at least the next decade, and probably much longer. A common data representation would reduce the complexity of these transformations (§3.3.).

3.1. Data Type Representations

Some of the issues involved in data type representations and transformations are illustrated by examining the primitive data types in four specialized processors: the Explorer II LISP processor, the TMS320C30 Digital Signal Processor, the MIPS R3000/4000 RISC Processor, and the MIL-STD-1750A Avionics Processor. All of these processors are likely choices for a military system. The Explorer II "LISP processor on a chip" was sponsored by DARPA for symbolic processing; the TMS320C30 is a recent digital signal processor with on-chip floating point hardware; the MIPS R3000/4000 RISC processor is one of two contenders for the new DoD standard 32-bit avionics processor (Common Avionics Processor or CAP-32); and the MIL-STD-1750A is the Air Force standard 16-bit avionics processor.

3.1.1. Byte Order: Big- or Little-Endian

In addition to the various numerical methods used to represent primitive data types, the byte order may differ when data is stored in memory (as discussed in §2.2). For example, in the Explorer LX system, both the Explorer LISP processor and the Motorola 68020 use the IEEE Floating Point Standard, but they store the bytes in memory in a different order [ExLX87]. This change in order makes the data from one processor unintelligible to the other. Data type transformations must take into account this difference in order. A recommended common data representation is presented in §3.3.4.

The transformation problem has been somewhat simplified within the last two years by the introduction of processors which can read data from memory using either the Big-Endian (BE) or Little-Endian (LE) byte ordering. Processors which have this capability include the Advanced Micro Devices 29000 [AMD89], the MIPS R3000 [MIPS88], the Motorola MC88100 [M88K 89], the Intel 80960CA [i960 89]. Not all of these processors can reconfigure their byte ordering "on the fly"; the MC88100 can, whereas the R3000 must be configured at startup. The Intel 80486 has a single cycle instruction for converting data between BE and LE [i486 89] [i486 90]. This is an example of augmenting the instruction set to include transformation instructions.

3.2. Data Type Transformations

This section provides background information on sharing of data, describes the information required to perform transformations, and categorizes the different types of transformations.

In order to maximize performance, only shared variables need to be transformed. This research focuses on transforming primitive data types rather than data structures. There are a wide variety of data structures, and it may be inefficient and difficult to duplicate an entire data structure on a dissimilar language/processor.

Shared data may be transferred between processors either through a common shared memory or by passing messages, usually across a network. There is some overlap between these two approaches as some message passing systems use a shared memory [HwBr84]. Generally, shared memory systems provide higher bandwidth between processors than message passing and are better suited for systems with a high degree of interaction between processors. A transformation is necessary when the shared memory is accessed by a processor which represents data differently than the representation used for shared memory. If a heterogeneous system had little interaction between processors, a message passing system might be suitable. In a message passing system, a transformation is necessary only when sending data to a dissimilar processor.

In order to transfer data between dissimilar processors, three items of information are necessary besides the data itself. These items are: (1) the source processor type, (2) the source data type, (3) the destination processor type, and (4) the destination data type. The source processor and data type are needed to correctly interpret the meaning of the data bits, while the destination processor and data type are needed to select the proper transformation. Some of these items may not be explicitly required if they are known implicitly.

3.2.1. Types of Transformations

The types of transformations can be divided into four categories, based on the data type and processor type (Figure 3-1). For the purpose of this categorization, we only consider whether the data type and processor type at each end of the transformation are the same or different. Examples of data types are integer, float, character, etc. Examples of processor types are R3000, MIL-STD-1750A, TMS320C30, etc. Each of the four categories of transformation can be further subdivided by considering the number of bits, order of the bits, and coding method (Figure 3-2). Coding method refers to choices such as two's complement, sign-magnitude, exponent bias, hidden bit, etc.

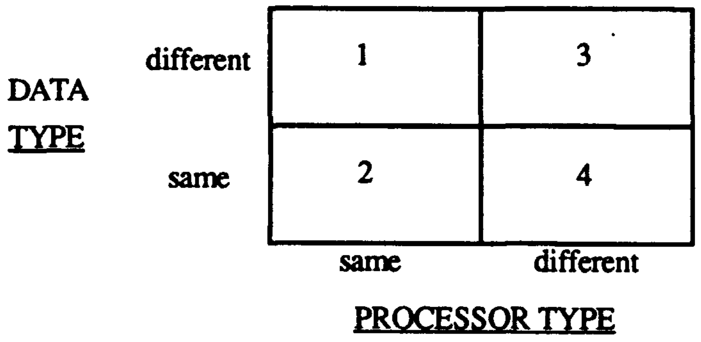
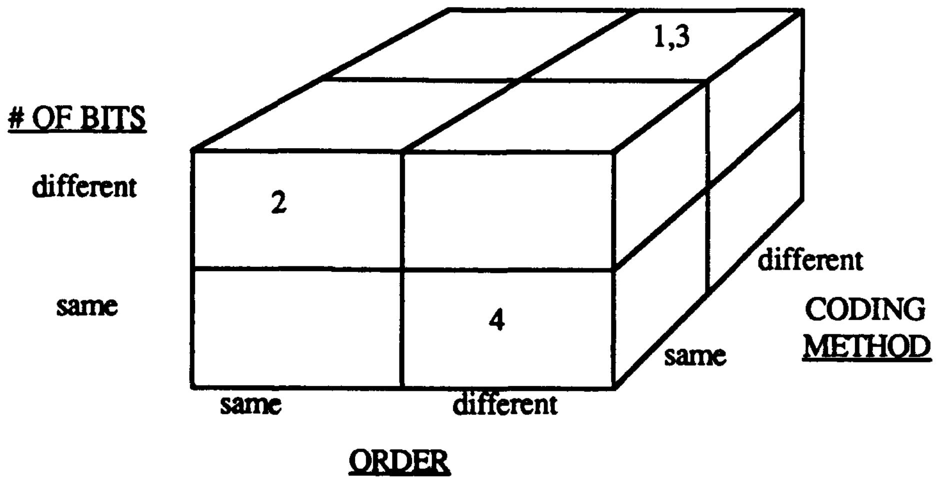


Figure 3-1: Types of transformations



Examples:
1. LISP BIGNUM to LISP 32-bit float
2. MIL-STD-1750A 48-bit float to MIL-STD 32-bit float
3. LISP BIGNUM to MIL-STD 32-bit float
4. TMS320C30 32-bit float to MIL-STD 32-bit float

Figure 3-2: Types of transformations(cont.)

The numbers in the blocks of Figure 3-1 and 3-2 correspond to the four examples listed at the bottom of Figure 3-2. In Example 1, the transformation of a LISP (i.e., the Explorer II LISP processor) BIGNUM (an integer which can be many words long) to a LISP 32-bit float falls under the category of 'same processor type/different data type' in Figure 3-1. This transformation can be further categorized in Figure 3-2 as 'different number of bits/different order/different coding method.' In Example 2, the only difference between the MIL-STD-1750A 48-bit float and the 32-bit float is the fact that the 48-bit float has an extra 16 bits of precision in the mantissa added to the end of the 32-bit float. So this transformation is categorized as 'same processor type/same data type/different number of bits/same order/same coding method.' Example 3 is similar to example 1, but is a 'different processor type.' In Example 4, the TMS320C30 puts the exponent at the high end of the word, while the MIL-STD-1750A puts the exponent at the low end of the word. So it is 'different order.' But they both have the same number of bits and use a two's complement coding method.

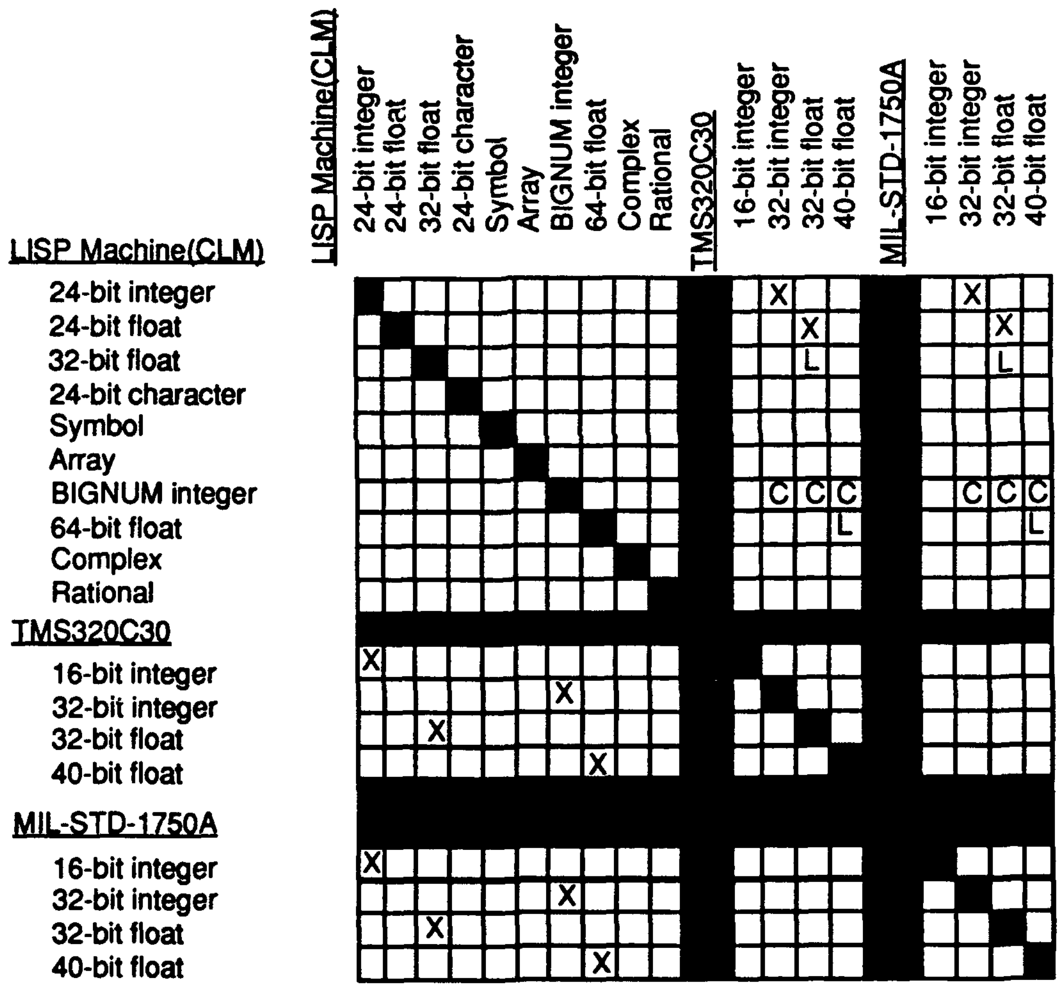
The exact set of transformations required will be dependent on the language and compiler used by each processor. The mapping between the primitive data types of each language and the primitive data types of the processor is the most significant factor in determining the transformations required. A possible set of transformations is shown in Figure 3-3 for three of the example processors (LISP, TMS320C30, MIL-STD-1750A). The set consists primarily of transformations between integers and floating point numbers of various bit lengths and representations. The integer transformations are straightforward and well understood. The most complex of the transformations are between the IEEE floating point standard and other floating point representations such as two's complement. This floating point transformation was coded in LISP in order to clarify the issues involved, and to determine the complexity of a hardware transformation unit (Appendix A). Floating point transformations are discussed in detail in §3.4.

Transformations to data types with fewer bits can cause a loss of precision and/or range. Selection of the proper destination data type is important, but this loss of information will be unavoidable in some cases. When this occurs, the transformation unit should set a flag indicating that information has been lost. Any subsequent exception processing would be handled by software.

Ideally, the data type of a shared variable would be agreed upon before any code is written. This would minimize loss of information resulting from transformations between different data types. But in practice, existing programs may sometimes be used rather than rewriting the program. Data type differences that arise from patching together existing programs should be handled by the transformation units.

In many cases, existing compilers can provide transformations between data types within a given processor (same processor type/different data type). These transformations have various names: implicit conversion, coercion, promotion, widening, cast, and explicit conversion. Some compilers (i.e., C, LISP) automatically perform implicit conversions when they encounter expressions containing mixed data types. Similarly, a programmer can specify explicit conversions to transform the type of a variable. (To aid in the detection of data type errors, Ada does not perform implicit conversions. All conversions must be explicit.) Some compilers use no-loss transformation rules when performing these conversions.

This, however, may not always be possible in a heterogeneous environment. When performing transformations across data and processor types (i.e., Processor A integer to Processor B float), it may be more efficient to first have the compiler match data types (Processor A integer to Processor A float), and then transform similar data types across processors (Processor A float to Processor B float).



	LISP Machine(CLM)										TMS320C30				MIL-STD-1750A			
	24-bit integer	24-bit float	32-bit float	24-bit character	Symbol	Array	BIGNUM integer	64-bit float	Complex	Rational	16-bit integer	32-bit integer	32-bit float	40-bit float	16-bit integer	32-bit integer	32-bit float	40-bit float
LISP Machine(CLM)																		
24-bit integer	■											X				X		
24-bit float		■											X				X	
32-bit float			■										L				L	
24-bit character				■														
Symbol					■													
Array						■												
BIGNUM integer							■					C	C	C		C	C	C
64-bit float								■						L				L
Complex									■									
Rational										■								
TMS320C30																		
16-bit integer	X										■							
32-bit integer							X					■						
32-bit float			X										■					
40-bit float								X						■				
MIL-STD-1750A																		
16-bit integer	X														■			
32-bit integer							X									■		
32-bit float			X														■	
40-bit float								X										■

Legend: X - Recommended transform C - May lose precision (Conditional) L - Loss of precision

Figure 3-3: A set of example transformations between data types.

3.3. A Common Data Representation

This section briefly reviews prior research on common data representations, examines three existing common data representations – Sun XDR, Cronus, and ASN.1 – and then proposes a recommended common data representation. The Sun XDR protocol is widely used between networked computers and is well documented. Cronus is less well known, but it is being developed with the support of the Air Force, and documentation on the data types was available. ASN.1 is part of the OSI international standard for computer communication.

The complexity of the transformation process can be reduced by storing shared data in a common, or standardized, data representation. A common data representation reduces the complexity of the transformations from Order(N^2) to Order(N), as shown in Table 4-1. The problems and errors encountered with transformations will vary with the common data representation that is chosen. A judicious choice of representations will minimize the complexity and cost of the hardware transformation units, reduce errors caused by transformations, and maximize performance. Ideally, a common data representation should maximize conformance with commonly used data representations and with existing standards.

• **Common Data Representation** – a standardized method of storing primitive data types. Defines the size and order of the bits, bytes, and words for specified data types.

Much of the prior research on data transfer between heterogeneous processors has concentrated on distributed operating systems and network services. Three recent distributed operating systems are Mach/Matchmaker and Agora at CMU [Rash87] [Bisi87], V System at Stanford [Cher83], and Cronus at BBN [Dean87]. Sun has also addressed the transformation problem in the area of networking heterogeneous computers through their External Data Representation (XDR) standard [XDR86]. All of these projects have used software transformation methods, using either operating system calls or software subroutines. Early software solutions are often later implemented in hardware to improve performance [Dean87]. This research examines the use of hardware for transformations, and compares the performance of hardware and software transformation methods.

Bisiani briefly addresses the problem of data representation and transformation at the processor level by saying "...sharing cannot happen at such a low level of data representation, e.g. because of incompatible byte orderings or alignment requirements of different processors" [Bisi87]. This research shows that sharing is possible at the processor level of data representation.

3.3.1. Sun XDR

Sun Microsystems developed the External Data Representation (XDR) protocol specification to allow two dissimilar machines to exchange operands over a network despite differences in byte ordering, word length, floating point representation, and so on [ExLX86]. XDR is part of the Network File System (NFS), "which provides a transparent file service between machines of different manufacture. The NFS is implemented on a wide range of platforms and is quite popular as the de facto standard for file-sharing in the 80's, and most likely the 90's as well [Rose90]."

Computers which use this standard perform transformations ("filter") their own internal representation of data before sending it out on the network. The destination computer than transforms the data from the XDR representation to its own internal representation. Although Sun created XDR to aid in networking, the XDR protocol itself does not involve networking. XDR is only a data representation standard, independent of where that data might reside. ("By some strange coincidence, the XDR representation is identical to the hardware representation used by Sun [Rose90].")

The XDR manual [XDR86] states that "XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine." Note that a transformation is not theoretically required when sending data to an identical processor. However, the destination processor must know if the data is not in the XDR representation so that it does not attempt to "un-transform" the data. For this reason, in a general heterogeneous environment, the XDR representation would normally always be used to avoid confusion. The XDR standard assumes that the order of the bits in a byte is preserved across hardware boundaries. The least significant bit is transmitted first in a Little-Endian style in conformance to the Ethernet standard. All data types require a multiple of four bytes (or 32 bits) of data, with unused bytes set to zero. The bytes are numbered from 0 through n-1, and lower numbered bytes are transmitted first.

The XDR standard defines 14 data types. Of these, six are primitive data types and eight are compound, or "constructor" data types, which are composed of multiple data types (Table 3-2). While the XDR standard uses a Little-Endian order for the bits in a byte, it uses a Big-Endian order

XDR Data Types	
Primitive	Constructor
Integer	Enumeration
Unsigned integer	Boolean
Hyper Integer	Opaque
Hyper Unsigned	Byte string
Single Float	Fixed-size array
Double Float	Variable-size array
	Structures
	Discriminated union

Table 3-2: XDR data types

for bytes in a word. The most significant of the four bytes is designated “byte 0” and the least significant byte is designated “byte 3”. Likewise, the most significant bit of a single precision floating point number (i.e. the sign bit) is designated “bit 0” and the least significant bit of the fraction is designated “bit 31”.

Integers are represented by a 32-bit two's complement (TC) notation. Unsigned Integers (U) are represented by a 32-bit unsigned binary number. Hyper Integer is a 64-bit two's complement integer. Hyper Unsigned is a 64-bit unsigned binary number. Single and Double Precision floating point numbers are represented using the IEEE standard. The remaining eight data types are constructor data types.

Enumerations have the same representation as integers. For example, the three colors red, yellow, and blue could be described by an enumerated type named “colors":

typedef enum {RED=2, YELLOW=3, BLUE=5} colors;

Actually, enumerations are better viewed as a class of data types rather than an individual data type. Since they are user definable, they would be difficult to implement in hardware. Booleans are an enumeration with the form:

typedef enum {FALSE=0, TRUE=1} boolean;

Opaque data is a class of user-named data types that are passed between processors uninterpreted. A counted byte string consists of an unsigned integer showing the number of bytes in the string, followed by the “n” bytes of the string. If “n” is not a multiple of four, the “n” bytes are followed by enough zero-valued bytes to make the total byte count a multiple of four.

Users can declare fixed-sized arrays of homogeneous elements. The elements are sent in their natural order, 0 through n-1. Variable-size arrays of homogeneous elements are preceded by an unsigned integer which specifies the size of the array. Structures are a class of constructor data types which can be defined by the user. The final data type, discriminated unions, provides a way of choosing from among a set of specified data types.

3.3.2. Cronus

Cronus is a distributed operating system developed at BBN Laboratories and supported by the Air Force's Rome Air Development Center. Like XDR, Cronus is a proprietary standard developed in the early 80's for exchanging data across heterogeneous networks. Both XDR and Cronus have a similar set of data types. The system-supplied common, or canonical, data types in Cronus include 16- and 32-bit two's complement integers, 16- and 32-bit unsigned integers, 32- and 64-bit IEEE floating point numbers, boolean, ASCII, and arbitrary strings of bytes [Dean87] [Scha88]. Cronus also has several other data types for system functions such as date, time intervals, host numbers, access control and unique numbers. New composite data types can be built out of the system-supplied data types [Dean87].

3.3.3. ASN.1

Abstract Syntax Notation One (ASN.1) is part of an evolving international standard which, although predicted to be very popular for network communications, is not an efficient common data representation for shared memory. It is mentioned here because in the future it is expected to have a significant impact on communication between heterogeneous computers across networks.

ASN.1 is a formal description language used to define data types. Within ASN.1 are a set of Basic Encoding Rules (BER) that describe one particular set of data types. Both the ASN.1 and the BER are based on the 1984 CCITT

Recommendation X.409. ASN.1 and the BER are part of a larger suite of protocols called the Open Systems Interconnection (OSI) Reference Model. "Although there are very few OSI systems deployed today, it is widely predicted that OSI will become the networking solution of choice within the next several years ... ASN.1 is destined to become the network programming language of the 90's, just as the C programming language is largely seen as having been the systems programming language of the 80's [Rose90]."

> The BER provides an encoding scheme general enough to support all machine architectures; however, the scheme is not particularly efficient to implement on any existing machine architecture. As such, the price of interoperability can be tremendous inefficiency. For example, informal measurements comparing the BER to [XDR] show that using the BER to encode an integer on a Motorola 68020 processor is anywhere from three to twenty times slower than using the XDR ... This has led to a controversy regarding the cost of using ASN.1 and the BER [Rose90].

Although ASN.1 and the BER are much more flexible, XDR is much closer to the representation actually used by processors. XDR thus allows simpler hardware transformations and is a better choice for a common data representation in shared memory.

3.3.4. Recommended Representation

This section proposes a common data representation for exchanging data between heterogeneous computers in hardware. This representation is a subset of XDR, and differs only in that the proposed bit and byte order is more consistent than XDR. However, the particular order which is chosen is not as important as simply agreeing on an order. From a practical standpoint, given the widespread acceptance of XDR, it would be reasonable to use that (inconsistent) order.

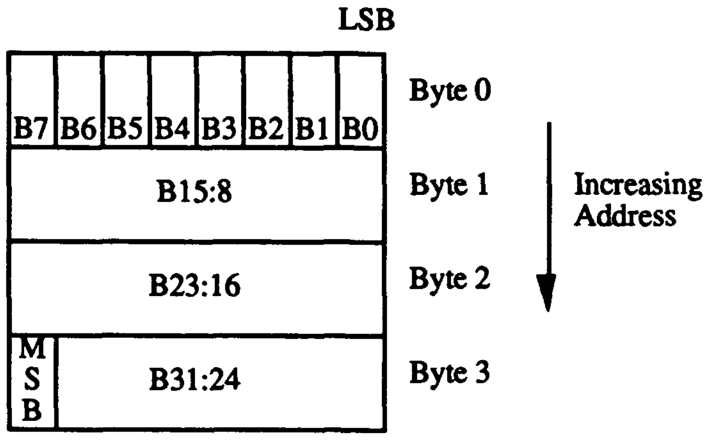
The number of primitive data types defined has been kept to a minimum to keep the hardware simple. The recommended representation has five primitive data types: single and double precision floating point, 32-bit signed integers, character, and opaque (or uninterpreted) data. Opaque is used when no transformation is required. Additional data types could be added if their frequency of use justified the additional complexity. Potential additional data types include 16-bit signed and unsigned integers, 32-bit unsigned integers, boolean, and one dimensional arrays (or vectors). A separate data type is not required for 16-bit integers since they can be represented in a 32-bit integer. The only reason to provide a 16-bit integer data type is to avoid the need to check for overflow when converting from a 32-bit integer.

The clear choice of representation for floating point numbers is the IEEE Floating Point Standard [IEEE85]. The increasing number of processors which use this representation will require no transformation, and some of those that do not use it provide instructions or routines for transforming their representation to and from the standard. The hidden bit provides one additional bit of precision over representations without a hidden bit, and values for singularity conditions are defined.

Signed integers are represented using a 32-bit two's complement notation. Two's complement is the most common representation used today for signed integers.

The character data type has four 8-bit ASCII characters packed in a 32-bit word. Unused bytes are set to zero.

The recommended order for data is a completely consistent Little-Endian for bits, bytes, and words (32-bit) (Figure 3-4) [Kirr83].



B=bit
LSB=Least Significant Bit
MSB=Most Significant Bit

Figure 3-4: Consistent Little-Endian order

Of the four example processors described in §3.1, the data paths on three are 32 bits wide, and one is 16 bits wide. As a result, the transformation units, shared memory, and the bus are all 32 bits wide in order to optimize performance for the 32-bit processors. The 16-bit processor requires a 16-bit buffer to capture the data from the 32-bit bus without delaying the bus cycle.

All data must be fully aligned. For a byte-addressable processor, the two least significant bits of the address must be equal to zero to be aligned on 32-bit word boundaries. This prevents data from extending across two words in memory. Data which is not aligned requires two memory cycles and shift operations before it can be used by the processor. Unaligned data was more common in the past when memory was more expensive, because it avoids unused bytes in memory. Many RISC processors today require alignment, preferring to waste bytes of memory rather than waste cycles aligning the data. Both the R3000 and the MC88100 require aligned memory addresses. Full alignment also eliminates address incoherency between Big-Endian and Little-Endian processors (§2.2).

The integer and floating point representation methods for the four example processors, and the three common data representations described in this section are summarized in Table 3-3.

Table 3-3: Integer and float representation methods

Processor	Integer size	Integer represent.	Float size	Float represent.	Byte Order
R3000	32	TC	32,64	IEEE	BE, LE
Explorer	32,64,∞	TC	32,64	IEEE	LE
TMS320	16,32	TC,U	16,32,40	TC(HB)	LE
1750A	16,32	TC	32,40	TC	BE
XDR	32,64	TC	32,64	IEEE	BE
Cronus	16,32	TC,U	32,64	IEEE	BE
Recomm.	32	TC	32,64	IEEE	LE

TC	Two's Complement	BE	Big-Endian
U	Unsigned integer	LE	Little-Endian
HB	Hidden Bit		

3.4. Floating Point Transformations

This section analyzes the errors caused by the transformation of 32-bit (single-precision) floating point numbers. The transformation of 64-bit, double-precision floating point numbers is virtually identical. Computers using the IEEE Floating Point Standard will not require a transformation and will not induce errors. Transformations between most other representations will lose precision and/or range.

D.W. Matula considered the effect of transformations on precision. He examined the number of digits required in different representation systems for one-to-one and onto mappings [Matu67]. He also examined the accumulated error of successive conversions using both rounding and truncation [Matu68] [Matu70]. Matula's work considered conversions between floating point numbers using differing radices. In today's computing environment, all computers known to the author use a radix of two for representing floating point numbers. The analysis in this section will use a radix of two as this more specific analysis will be more useful to today's computer designers. The analysis could be generalized to consider other radices.

3.4.1. Transformation Mappings

When examining transformations between different representations, it is useful to consider whether they are one-to-one or onto mappings [Matu67].

With a one-to-one (1:1) mapping, each element of the domain is mapped to a different element of the range (Figure 3-5). A mapping with this property makes it possible to input two numbers that are very close to each other and yet still have them remain distinct from one another in the new representation. There may be elements in the range which have no corresponding element in the domain (i.e., the range has more elements than the domain).

With an onto mapping, each element of the range is mapped to by at least one element of the domain. In an onto mapping, there may be elements in the domain which have no corresponding element in the range. Two elements in the domain can map to one element in the range.

If a mapping is both one-to-one and onto, it is a one-to-one correspondence (Figure 3-5). This is true because the only mapping that satisfies both one-to-one and onto requirements is a mapping with an equal number of elements in both the domain and range which are uniquely paired.

The first concern when performing a transformation to another representation is whether that transformation is one-to-one. As long as the destination (or intermediate) representation has an equal or greater number of bits than the source representation, the transformation is one-to-one and no information will be lost in the transformation. At this point, the fact that there may be extra elements in the intermediate representation is not a concern. A given value in the source representation can be transformed into the intermediate representation and then transformed back with no loss of information.

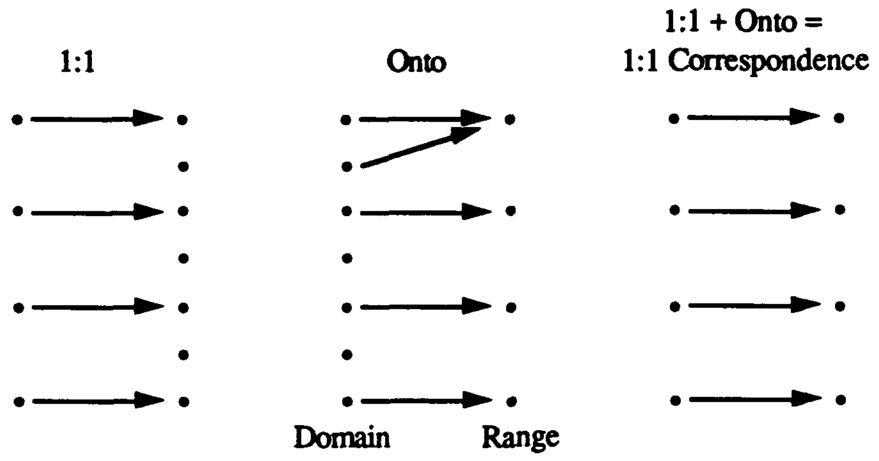


Figure 3-5: Mappings: one-to-one, onto, and one-to-one correspondence

However, the presence of extra elements in the intermediate representation means that information may be lost when transforming to a representation with fewer bits. This is an onto mapping. This can occur when an operation is performed on a value in the intermediate representation. The resulting value in the intermediate representation may not have a unique corresponding element in the source representation. In other words, multiple values in the intermediate representation may map to the same value in the source representation. A rounding or truncation transformation must be used to map values from the intermediate representation back to the source representation. Multiple conversions between representations using incommensurable bases can cause errors to build up [Matu70]. The bases P and Q are commensurable if and only if $P^i = Q^j$ for some nonzero integers i and j. For example two, four, eight, and sixteen are commensurable bases, whereas base ten is incommensurable with these bases. The major lesson to learned from Matula's work is to avoid multiple conversions in-and-out of base ten. Programs requiring decimal input and output may have errors at the start and end, but programs do not generally require intermediate transformations to base ten.

3.4.2. Floating Point Transformation Errors

This section examines the errors that occur when transforming radix-two floating point numbers between different representations. Errors can occur due to differences in five areas:

1) the number of bits in the exponent
2) the number of bits in the mantissa
3) the singularity conditions
4) the representation method
5) the rounding method

These errors will be illustrated by using as an example two of the most common floating point representations in use today – the IEEE Floating Point Standard (IEEE) and two's complement (TC) without a hidden bit. The Explorer LISP processor uses the IEEE representation, while the TMS320C30 and the MIL-STD-1750A each use slightly different two's complement representations (the exponent and mantissa are reversed in order and the C30 uses a hidden bit). The mapping between these two representations is illustrated in Figure 3-6 and described in the following sections.

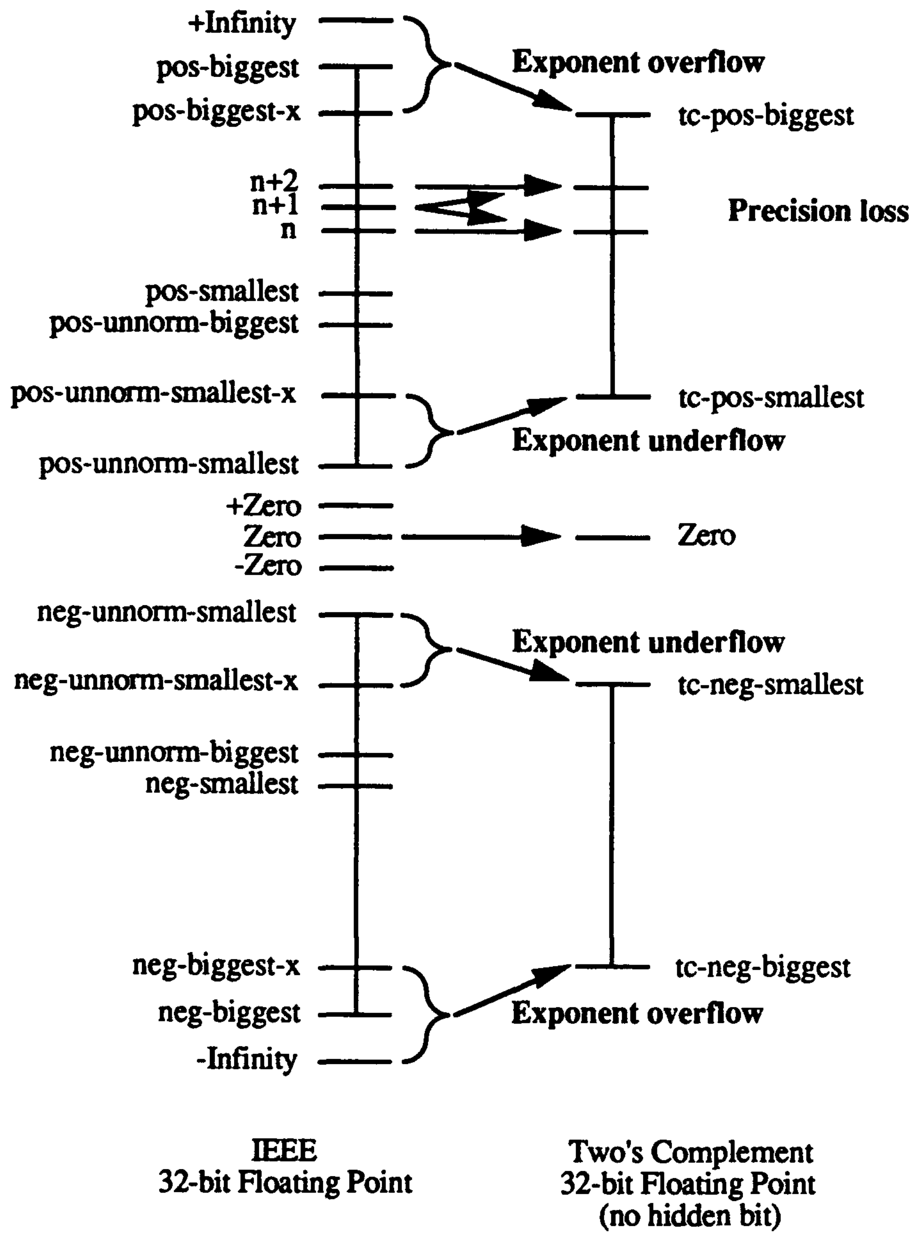


Figure 3-6: Mapping between IEEE and two's complement floating point

3.4.2.1. Exponents

Both representations use an 8-bit exponent. The IEEE exponent is biased by +127 and could represent numbers from -127 (00000000) to +128 (11111111). But these two extreme values, -127 and +128, are reserved for indicating denormalized numbers and the quasi-infinites, respectively. So the usable range is -126 to +127. The two's complement exponent represents numbers from -128 (10000000) to +127 (01111111). The values illustrated in Figure 3-6 are shown in Tables 3-4 and 3-5 in decimal and binary, respectively.

Table 3-4: Decimal representation of values in Figure 3-6

Name (IEEE vectors)	IEEE Value[e]	TC Conversion Value	Flags
pos-infinity	NaN	1.9999998 * 2**126[a]	I,O
pos-biggest	1.9999999 * 2**127	1.9999998 * 2**126[a]	O,P
pos-biggest-x	1.9999999 * 2**126	1.9999998 * 2**126[a]	P
pos-smallest	1.0 * 2**-126	1.0 * 2**-126	
pos-unnorm-biggest	0.9999999 * 2**-126	0.9999999 * 2**-126	
pos-unnorm-smallest-x	0.125 * 2**-126	0.125 * 2**-126[b]	
pos-unnorm-smallest	0.00000012 * 2**-126	0.125 * 2**-126[b]	U
pos-zero	0 * 2**0	0 * 2**0	
zero	0 * 2**0	0 * 2**0	
neg-zero	-0 * 2**0	0 * 2**0	
neg-unnorm-smallest	-0.00000012 *2**-126	-0.12500012*2**-126[c]	U
neg-unnorm-smallest-x	-0.12500012 *2**-126	-0.12500012*2**-126[c]	
neg-unnorm-biggest	-0.9999999 * 2**-126	-0.9999999 * 2**-126	
neg-smallest	-1.0 * 2**-126	-1.0 * 2**-126	
neg-biggest-x	-1.0 * 2**127	-1.0 * 2**127[d]	
neg-biggest	-1.9999999 * 2**127	-1.0 * 2**127[d]	O,P
neg-infinity	NaN	-1.0 * 2**127[d]	I,O

<u>Comments</u>:
[a]tc-pos-biggest
[b]tc-pos-smallest
[c]tc-neg-smallest
[d]tc-neg-biggest
[e]Unbiased exponent; add 127 for biased value

<u>Flags</u>:
O - Overflow
U - Underflow
I - Infinity
P - Precision loss

Table 3-5: Binary representation of mantissa values in Figure 3-6

Name (IEEE vectors)	IEEE Value[e,f]	TC Conversion Value[g]	Flags
pos-infinity	X.X...X * 2**128	0.1...1 * 2**127[a]	I,O
pos-biggest	1.1...1 * 2**127	0.1...1 * 2**127[a]	O,P
pos-biggest-x	1.1...1 * 2**126	0.1...1 * 2**127[a]	P
pos-smallest	1.0...0 * 2**-126	0.10...0 * 2**-125	
pos-unnorm-biggest	0.1...1 * 2**-126	0.1...1 * 2**-126	
pos-unnorm-smallest-x	0.001 * 2**-126	0.1 * 2**-128[b]	
pos-unnorm-smallest	0.0...01 * 2**-126	0.1 * 2**-128[b]	U
pos-zero	0.0...0 * 2**0	0.0...0 * 2**0	
zero	0.0...0 * 2**0	0.0...0 * 2**0	
neg-zero	-0.0...0 * 2**0	0.0...0 * 2**0	
neg-unnorm-smallest	-0.0...01 * 2**-126	-0.01...100*2**-126[c]	U
neg-unnorm-smallest-x	-0.0010...01*2**-126	-0.01...100*2**-126[c]	
neg-unnorm-biggest	-0.1...1 * 2**-126	-0.0...01 * 2**-126	
neg-smallest	-1.0...0 * 2**-126	-0.10...0 * 2**-125	
neg-biggest-x	-1.0...0 * 2**127	-0.0...0 * 2**127[d]	
neg-biggest	-1.1...1 * 2**127	-0.0...0 * 2**127[d]	O,P
neg-infinity	X.X...X * 2**-128	-0.0...0 * 2**127[d]	I,O

Comments:
[a]tc-pos-biggest
[b]tc-pos-smallest
[c]tc-neg-smallest
[d]tc-neg-biggest ($-0.0...0_{2\,tc} = -1.0_{10}$)
[e]Unbiased exponent; add 127 (01111111_2)for biased value
[f]Mantissa base 2, hidden bit shown (i.e. 1.XXX)
[g]Mantissa base 2, two's complement

Flags:
O - Overflow
U - Underflow
I - Infinity
P - Precision loss

3.4.2.2. Mantissas

Both representations use 24-bits for the mantissa (sometimes also called the characteristic). However, the IEEE mantissa uses a hidden bit to gain an extra bit of precision. The mantissa is normalized to a sign-magnitude binary number between one and two (i.e., 1.011). The 1 to the left of the binary point is not represented – only the fractional part is represented. The mantissa sign bit is in bit 31. The TC mantissa uses a normalized TC fractional representation. The IEEE mantissa with the hidden bit can represent twice as many numbers as the two's complement mantissa without the hidden bit (2^{24} vs 2^{23}).

3.4.2.3. Singularity Conditions

The three floating point singularities, or abnormal situations, are exponent overflow, exponent underflow, and Order-of-Magnitude Zero (OMZ) [Hwan79]. The first singularity, exponent overflow, occurs when the exponent is greater than the largest allowable positive value. The result can be positive or negative and is represented by the symbols $+\infty$ and $-\infty$; these are called quasi-infinites. The second singularity, exponent underflow, occurs when the exponent is less than the most negative allowable value. The result is represented by $+\varepsilon$ and $-\varepsilon$ which are called infinitesimals. The final singularity, Order-of-Magnitude Zero (OMZ) occurs when a floating point number has a zero mantissa, while the exponent can assume any legitimate value. As shown in Table 3-1, most representations do not allow OMZs, but require that True Zero be represented with a zero exponent as well as a zero mantissa. Only the VAX permits OMZ.

The IEEE representation reserves values for indicating exponent overflow and underflow. In addition, the use of unnormalized numbers allows gradual underflow by sacrificing the number of bits of precision.

Problems will arise in the following situations when attempting to perform conversions between the two representations:

1. When the IEEE exponent equals 127 or 128 (i.e. E=11111110 or 11111111) an overflow will occur when converting to TC. An overflow must be signaled to the TC processor. For example:

$1.11...1_2 \times 2^{126}$ (IEEE) —> $0.1...1_2 \times 2^{127}$ (TC OK)

$1.0_2 \times 2^{127}$ (IEEE) —> $0.1_2 \times 2^{128}$ (TC overflow)

2. When the TC exponent equals -126, -127 or -128 (10000010, 10000001 or 10000000), underflow will occur when converting to IEEE. The conversion can be made to an IEEE unnormalized number. However zero, one, or two bits, respectively, of precision will be lost. Similarly, if a positive unnormalized IEEE number less than ($2^{-3} \times 2^{-126} = 2^{-129}$) is converted, an underflow will occur. If a negative unnormalized IEEE number less than or equal to -2^{-129} is converted, underflow will occur.

3. A normalized IEEE number with the LSB=1 will lose one bit of precision in the conversion to TC, whether rounding or truncation is used.

This section has described the errors that can occur in the transformation between the IEEE floating point standard and a two's complement representation. One bit of precision may be lost due to the extra hidden bit in the IEEE mantissa. Exponent overflow and underflow can occur for numbers which are near the maximum and minimum limits of the representation.

Chapter 4

Alternative Transformation Approaches

This chapter examines alternative locations and methods for performing data type transformations. Local, central, and distributed locations are compared in terms of complexity and concurrency. A spectrum of implementations is then considered, ranging from software subroutines on the host processor to combinatorial logic. The general models presented in §4.2 were developed early in this research and assume a message passing approach. These models provided the basis for the decision to consider in more detail the shared memory approach with local hardware transformation units which is discussed in Chapter 5.

4.1. Location of Transformation Units

Data type transformations can be performed locally, centrally, or can be distributed throughout the system. These three approaches are briefly defined below and are explained in more detail in the following sections.

Local – the transformations are performed by the processor itself, or by a transformation unit (TU) attached to each processor.

Central – all transformations are performed by a central TU.

Distributed – the transformations are performed by specialized TUs which are distributed throughout the system.

The following twelve questions will be used to evaluate these three locations. This analysis assumes a message-passing approach is used.

(1)Will bottlenecks occur if additional processors are added (non-inductive) [LiMa87]?

(2) How many transformation units are required?

(3) What is the minimum complexity of the transformation unit?

(4) What is the maximum complexity of the transformation unit?

(5) What is the minimum complexity of the sum of all the transformation units in the system?

(6) What is the maximum complexity of the sum of all the transformation units in the system?

(7) What is the maximum concurrency (i.e. how many transformations can be performed at one time?)

(8) How many additional transformation units are required to add a processor if that type is already present?

(9) How many additional transformation units are required to add a new processor type?

(10) What is the performance of this transformation scheme (i.e. the relative transformation time)?

(11) What information is required by the transformation unit (i.e. source processor type, data type, destination processor type)?

(12) Does this transformation scheme require specialized control?

4.1.1. Local Transformation

With the local transformation approach, the transformation is performed by the processor itself, or by a TU attached to each processor. Data is transformed to a common representation and sent to the destination TU. The destination TU transforms the common representation into the destination representation. Some processors may not require a separate TU if they already have specialized hardware for shifting bits. For example, the LISP processor has a barrel shifter for extracting tag bits. If this hardware can be used effectively, a separate TU may not be required. This is an example of an augmented instruction set (§4.3).

The local transformation approach requires a representation which is common to all the TUs. The 'network' side of all the TUs uses this common representation. Each TU translates between the common representation and the representation on its particular processor. The common representation must be capable of representing all data types which are to be transformed, preferably without loss of precision. The data type description is needed by the source and destination TUs so the appropriate transformation can be performed. The LISP processor uses a tagged data representation, which includes a description of the data type with the data.

The local transformation approach will not cause bottlenecks as additional processors are added. Since each processor has a TU, the number of TUs grows with the number of processors in the system. For P processors, P TUs are required. The minimum possible complexity of each TU is based on the assumption that each data type in a given processor (P_x) is transformed to only one data type in the common representation. This is the lower bound on the complexity, or number of transformations required, of each TU. So if n_x is the number of data types for processor P_x, $2n_x$ transformations are required to transform these data types to and from the common representation. Some data types may require more than one transformation. For example, an integer may require transformations to integer, character, or float depending on the situation. The maximum possible complexity is based on the assumption that each data type in a processor is transformed to all data types in the common representation. This is the upper bound on the number of transformations required. If n_c is the number of data types in the common representation, $2n_x n_c$ is the upper bound on the number of transformations required for P_x. The minimum total system complexity for all processors is:

$$\sum_{i=1}^{P} (2n_x) = 2N_P, \ (N_P = n_1 + n_2 + \ldots + n_P)$$

where N_P is the total number of data types in all processors.

Likewise, the maximum total system complexity is $2N_P n_c$. The maximum concurrency is P since each of the P TUs can work simultaneously.

One additional TU is required when adding either an additional processor of a type already present in the system, or a new processor type. In the former case, the TU will be a copy of an existing unit. In the latter case, a new TU must be designed. In either case, existing TUs are not affected.

The local scheme requires two transformations. The data is transformed from the source representation to the common representation, and then from the common representation to the destination represe· .ion. If T is the time for one transformation, the total transformation time is 2T. The transformation time is discussed in more detail in §4.2.

The only information the TU needs is the data type description. The source processor type and destination processor type are not needed. The source (destination) processor type is implicitly known by each source (destination) TU, and the destination (source) processor type is not needed since the data is transformed to (from) a common representation. However, the interconnection network still needs to know the destination. No specialized control is required.

4.1.2. Central Transformation

In the central transformation scheme, all transformations are performed by a central TU. Data is sent in the source representation to the central unit, transformed to the destination representation, and sent on to the destination.

The central TU can become a bottleneck. As additional processors are added, a central unit of a given capacity will become saturated. Multiple central units could be provided, but they would require some means of distributing the load. This section assumes one central unit. The complexity is based on the same assumptions discussed under the local transformation scheme. In addition, it is assumed that the central TU uses a common internal representation. This reduces the number of transformations required from Order(Q^2) to Order(Q), but at the time expense of two transformations.

Since the central transformation approach has only one unit, the total system complexity is the same as the unit complexity. The total system complexity of the central approach will be less than the distributed approach when the number of processors is greater than the number of processor types ($P > Q$); in other words, when there are multiple copies of some processor types. But as before, since two transformations are required the transformation time is 2T. The maximum concurrency is one since there is only one TU (assuming no pipelining in the central TU). Additional processors of a type already present in the system do not require any changes in the central TU. However, the addition of a new processor type would require a new central TU.

With message passing, the source processor type, source data type, destination processor type and destination data type must all be explicitly communicated to the central unit. No additional control is required.

4.1.3. Distributed Transformation

In the distributed transformation scheme, the transformations are performed by specialized TUs which are distributed throughout the system. Each TU is only capable of performing transformations from one source processor type to one destination processor type. For Q processor types, Q^2 units would be required to provide all possible transformations. This number includes a TU for the case where source processor type and destination processor type are the same. This unit may be required for 'same processor type/different data type' transformations. Bottlenecks may occur as additional processors are added. The minimum unit complexity is n_x. The maximum unit complexity is $n_x n_d$, where n_d is the number of data types in the destination processor for that TU. The minimum total system complexity is:

$$Q(n_1 + n_2 + \ldots n_q) = QN_Q$$

The maximum total system complexity is:

$$\sum_{i=1}^{Q} \sum_{j=1}^{Q} n_i n_j = N_Q^2$$

This includes identity transformations. The maximum concurrency is Q^2 since there are Q^2 TUs. For P = Q, many of these units would be idle at any given time. Additional copies of existing processors do not require any additional TUs. New processor types require Q additional TUs. Only a single transformation is required in this approach, so the transformation time is T. Only the data type is required since the source and destination type are implicitly known. Additional control is required to direct the data to the appropriate TU. But since the destination processor must be communicated to the network anyway, this may not be a significant factor.

The number of TUs may be reduced by using a partially connected matrix [Caro86]. Rather than providing TUs between all processor types, only provide TUs between heavily used processor types. Less frequently used transformations can be accomplished by routing the data through more than one TU. For example, assume processor A has TUs to and from processors B, C, and D. A transformation between processors C and D can be accomplished via (C to A) and (A to D) transformations. This approach reduces the number of TUs from that required in the completely connected distributed scheme. But it requires additional control and multiple transformations.

4.1.4. Conclusions on TU Location

The answers to the twelve questions posed at the beginning of §4.1. are summarized in Table 4-1. The local transformation approach has the advantages of flexibility, moderate complexity, and ample concurrency. The addition of a new processor to a system does not affect the other processors and simply requires an additional TU. Each TU is relatively simple as it only has to transform data from that processor's representation to the common representation. Since each processor has it own TU, bottlenecks will not occur due to transformations. A central TU is complex, inflexible, and subject to bottlenecks. A distributed approach has high complexity and useless excess concurrency. For these reasons, the designs in Chapter 5 will use local TUs. Note however that these three approaches are points on a spectrum, and other combinations are possible. For example, multiple central units could be used to minimize bottlenecks, or a local transformation unit could be shared by two identical processors.

Table 4-1: Comparison of alternative locations

Question	Local	Central	Distributed
1. Bottlenecks	No	Yes	Yes
2. Units Required	P	1	Q^2
3. Minimum Unit Complexity	$2n_x$	$2N_Q$	n_x
4. Maximum Unit Complexity	$2n_x n_c$	$2N_Q n_c$	$n_x n_d$
5. Minimum System Complexity	$2N_P$	$2N_Q$	QN_Q
6. Maximum System Complexity	$2N_P n_c$	$2N_Q n_c$	N_Q^2
7. Maximum Concurrency	P	1	Q^2
8 Additional Processors	1	0	0
9. New Processor Types	1	1	Q
10. Transformation Time	2T	2T	T
11. Information Required	t	s/t/d	t
12. Control Required	No	No	Yes

P = number of processors
Q = number of processor types ($Q \leq P$)
N = total number of data types
$N_P = n_1 + n_2 + \ldots + n_P$
$N_Q = n_1 + n_2 + \ldots + n_Q$
n_x = number of data types in processor x
n_c = number of data types in common representation
n_d = number of data types in destination processor
T = transformation time
s/t/d = source processor type, data type, destination processor type

4.2. Performance Models for Alternative Locations

This section will derive network performance models for the three transformation locations using message passing. These models will show the average time required to transmit a value between any two processors ($T_{avg}X$).

4.2.1. Local Transformation

In the local transformation scheme, the time required to transmit a value from a given processor to another is:

$$(T_s + T_n + T_d)$$

T_s and T_d represent the average transformation time for the source and destination TUs. T_n is the time required to transmit the information through the interconnection network connecting the processors. The average transformation time is a weighted average of all the possible transforms. Just as different instructions require different amounts of time in a processor, likewise some transformations will have different execution times than others. The average transformation time for a given TU is:

$$\sum_{i=1}^{n} p_i t_i$$

There are n transforms, each having its own execution time, t_i, and probability, p_i. (Note: Lowercase p refers to a probability, uppercase P is the number of processors in the system.)

The network time can be broken down into access time (T_a) and transmit time (T_t).

$$T_n = T_a + T_t$$

T_a is the average time required to gain access to the network, and T_t is the time required to transmit the information across the network. The significance of these two components will vary with the type of network used. For a dedicated data bus between two processors, both of these times may be negligible. For a shared bus, the access time may be significant, while the transmit time is negligible. For a shared multistage network, both components may be significant.

The average time to transmit a value from any processor to any other processor will be the weighted average of all possible combinations:

$$T_{avgL} = \sum_{s=1}^{P} \sum_{d=1}^{P} p_{sd} (T_s + T_n + T_d), \text{ for } s \neq d$$

Here p_{sd} is the probability of transmitting a value from source s to destination d, and the sum of all the probabilities equals one. P is the number of processors in the system. When the source and destination are the same (s = d), the transmit time is zero since the value is already present in the source processor. Hence the above summation has the restriction that s≠d.

4.2.2. Central Transformation

In the central transformation scheme, the average time required to transmit a value from any processor to any other processor is:

$$T_{avgC} = (T_C + 2T_n)$$

T_C is the average time required by the Central TU and is a weighted average of all the possible transforms. Assuming a common internal representation:

$$T_C = \sum_{s=1}^{P} \sum_{d=1}^{P} p_{sd} (T_s + T_d)$$

The total number of transforms in the Central TU will be much higher than the number in any one of the local TUs. However, if a common internal representation is used in the Central TU, the total number of unique transforms in both the central and local approaches will be the same. The network time, T_n, is multiplied by two since two network access and transmit times are required; once from the source processor to the Central TU, and once from the Central TU to the destination processor.

4.2.3. Distributed Transformation

In the fully connected distributed transformation scheme, the time required to transmit a value from one specific processor to another is:

$(T_{sd} + 2T_n)$

T_{sd} is the average transformation time for the 'sd' TU connecting source processor, s, to a destination processor, d. Only one transformation is required. As in the central transformation model, two network accesses are required.

The average time to transmit a value from any processor to any other processor is (Df= Distributed, fully connected):

$$T_{avgDf} = \sum_{s=1}^{P} \sum_{d=1}^{P} p_{sd} (T_{sd} + 2T_n)$$

In the partially connected distributed transformation scheme, the time required to transmit a value between two processors that are not directly connected is:

$(T_{sa} + T_{ad} + 3T_n)$

This assumes that both the source and destination processor are connected via a common processor 'a'. An additional network access is required to go from the 'sa' TU to the 'ad' TU.

The average time required to transmit a value between any two indirectly connected processors in the distributed, partially connected scheme is (Dp= Distributed, partially connected):

$$T_{avgDp} = \sum_{s=1}^{P} \sum_{d=1}^{P} p_{sd} (T_{s1} + T_{12} + T_{23} + \ldots + T_{md} + (m+1)T_n)$$

where m is the number of TUs used.

The combined average time for both directly and indirectly connected processors is:

$$T_{avgDc} = (p_1)T_{avgDf} + (p_2)T_{avgDp}$$

where p_1 is the probability of a direct connection and p_2 is the probability of an indirect connection. The value of p_2 is likely to be small. If two processors communicate frequently it is probably worthwhile to have a dedicated TU, hence a direct connection.

The total number of TUs in the fully connected distributed scheme is Q^2 where Q is the number of processor types. In the partially connected scheme, the number of TUs can be reduced to 2Q-1 assuming we limit the number of sequential transforms to two.

4.3. Implementation Alternatives

A spectrum of implementation alternatives exist for transforming data types (Figure 4-1). The transformations could be performed by combinatorial logic, by a processor (or coprocessor) with specialized transformation instructions, by software on a processor with an augmented instruction set, or by network software, operating system calls or software subroutines on a general purpose processor. The type of implementation is a function of the cost and performance requirements. Each of these alternatives are discussed in this section.

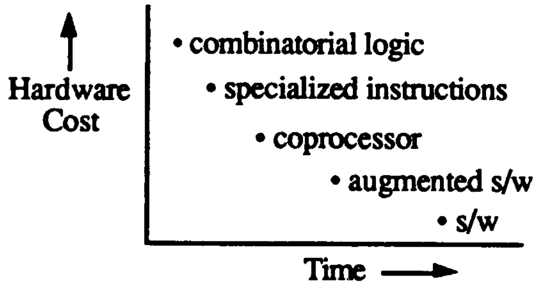


Figure 4-1: Spectrum of transformation implementation alternatives

4.3.1. Transformations with Combinatorial Logic

In order to demonstrate the performance advantages of hardware transformation units, a transformation unit using combinatorial logic was designed for one of the more difficult transformations. This unit performs transformations from 32-bit, single-precision IEEE floating point numbers to a two's complement floating point number. Four exception flags indicate that one of the following singularity conditions has occurred: exponent overflow or underflow, zero exponent, and infinity. These exceptions must then be handled in software.

The conceptual logic was developed by writing the transformation in LISP code (Appendix A). The hardware transformation unit was then designed using nine 20-pin PALS (PAL16L8). These parts were used due to their availability and have a maximum latency of 35 nanoseconds (ns). Two levels of chips are required to complement the mantissa which results in a total latency of 70 ns. This provides a transformation rate of over 14,000,000 transformations per second. A single-level design would cut the latency in half, but would require a 21 input OR gate.

Faster versions of this PAL are available with latencies as short as 10 ns (PAL16L8D). Two chip delays at 10 ns each results in a total latency of 20 ns, and a transformation rate of 50,000,000 transformations per second. The performance could be further improved and chip count reduced by using very large PALs (i.e., a 100 pin PAL) or ultimately a custom VLSI transformation unit.

If further improvements in performance are desired, pipelining could be used for speedup. The number of stages in the pipeline would be determined by the desired speedup, the length of the input vectors and cost constraints.

Texas Instruments USA has recently begun sampling a device very similar to the one just described that converts between the IEEE and the TMS320C30 floating point representations [Reif90]. This bidirectional custom gate array is already in use by TI Japan to reduce the latency in sharing data with graphics devices. The cost of this gate array is around ten dollars.

4.3.2. Transformations with Specialized Instructions

A transformation coprocessor is essentially a specialized 32-bit processor with a very specific instruction set tailored to transformations. It does not require complex hardware such as a multiply unit, and most of the instructions are fairly simple. Such a unit would be expected to be at the low end of the processor cost spectrum, even considering that the production volume would be much lower than a more general purpose processor.

Three instructions are necessary for transformations: shift, add/subtract, and two's complement. These instructions could be added to an instruction set for a new processor, or a transformation coprocessor could be designed which would allow users to define specific transformations. Such a device would be relatively simple and inexpensive compared to current microprocessors.

The shift operation is required for bit reordering and byte reversal. This is often implemented in hardware with a barrel shifter. However, even a barrel shifter requires multiple operations for each part of the word to be shifted. For example, the reversal of four bytes requires at least four operations. An $n*\log_2 n$ network can also perform bit reordering, although the flexibility of such a network will generally not be needed. The fastest implementation for a pure reordering transformation is a hard-wired connections between the appropriate input and output pins. A pure reordering transformation is one in which the bits are moved to a different position, but no arithmetic operations are required. The only pure reordering transformations for the four example processors occur with the IEEE floating point (Big-Endian to Little-Endian, and Explorer to Little-Endian).

Addition and subtraction are required for biasing and unbiasing exponents.

The two's complement operation is required for transformations between sign magnitude and two's complement. This operation complements all the bits (one's complement) and adds one. Hwang[Hwan79] describes a bit-scanning circuit for performing this operation.

Not all processors will require a transformation coprocessor. Processors which use Little-Endian, two's complement integers, and the IEEE Floating Point standard may not require any transformations. Some processors already have an instruction set which has been augmented with instructions that can be readily applied to transformations. These are discussed in the next section.

4.3.3. Transformations in Software

The alternative to a hardware transformation unit is to perform the transformations in software on the host processor. In order to accurately determine the performance advantage of a hardware transformation unit, the time required to perform the transformations in software must be determined. This time will vary depending on the processor and the instruction set. The TMS320C30 and the Explorer LISP processor both have barrel shifters which are very useful in performing transformations. Similarly, the MC88100 can reorder bytes "on the fly" between Little-Endian and Big-Endian byte ordering. So although these processors do not have specific instructions for performing transformations, they do have an augmented instruction set that can perform transformations more quickly than a general purpose processor.

The TMS320C30 User's Guide [TMS88] provides two versions of a program which converts floating point numbers between its own representation and the IEEE Standard. These programs provide excellent data on the best possible time required to perform a floating point transformation in software. The programs are hand-coded in assembly language on a digital signal processor with a powerful, single-cycle instruction set and a barrel shifter. The fast version of the program does not properly handle the special cases of denormalized numbers, infinity, and NaN (Not a Number), while the complete version handles all the special cases. The program length in words and the number of cycles required are shown in Table 4-2.

Table 4-2: Software transformation time

Transformation	Program Length (words)	Cycles
IEEE => C30	12 (34)	8/12 (12/23)
C30 => IEEE	15 (25)	10/14(11/31)

Cycle time = 60 ns
Fast version (Complete version)
Best/Worst case

In order to compare a hardware TU against software with similar capabilities, the best case time for the complete version of the program will be used. The best case time represents the time for the normal transformation. This is comparable to a hardware TU since it would probably trap to software for the special cases. This best case time is 11 or 12 cycles, depending on the direction of the transformation, and is shown in bold print in Table 4-1. Another 17 to 25 cycles are required for procedure call overhead, and pipeline fill time and conflicts, which yields a total time of around 35 cycles. The same program on other, less powerful processors would require additional instructions and cycles, perhaps 50 to 100 cycles. In contrast, the proposed hardware TU can perform a transformation in a single cycle (§5.2.2., Table 5-3).

4.3.4. Transformations in Heterogeneous Networks

This section compares shared memory and networks in regards to the time required to access and transform shared data. A heterogeneous shared memory multiprocessor provides the capability to access shared data two to three orders of magnitude faster than a heterogeneous network. Two of the factors which contribute to this difference in performance are the overhead of message passing across a network and software transformations. The published papers on the projects referred to in this section only provide the total time for shared memory access. They do not provide component times such as processor cycle time, or the relative contributions of the network and the software transformations.

For this research, the predicted time for shared memory access and transformation is 13.7 cycles (§5.3.1., Table 5-8). At a cycle time of 60 nanoseconds (16.67 Mhz), this would be 822 ns. The performance advantage of this approach relative to the following approaches is shown in Table 4-3.

In the Agora project [BiFo87], the write access time averages 217 microseconds (μsec) and the read access time averages 56 μsec. That same paper also provide values for sharing on general purpose systems:

> Moreover, in current implementations on general purpose systems, communication is rather expensive since there is a message passing overhead even on shared memory architectures (currently about 2 ms for a general purpose 1 MIPS machine) [BiFo87].

The Nectar project [Arno89] has as a performance goal the ability to send a message between processors in under 30 μsec.

Table 4-3: Shared data access time comparison†

Project	Access Time	Slower by factor of:
This research	822 nsec (13.7 cycles)	–
Agora write	217 μsec	264
Agora read	56 μsec	68
General Purpose	2 ms	2433
Nectar	30 μsec	36

†comparison of total access time; component times such as processor cycle time not available

4.4. Conclusions on Transformation Alternatives

This chapter has shown that the use of shared memory and hardware transformation units can provide a significant performance advantage over message passing across networks and software transformations. Table 4-4 compares the transformation time of the implementation alternatives discussed in this section. The second column list the number of cycles required to perform a transformation, the third column lists the cycle time, and the fourth column lists the transformation time which is the product of the previous two columns.

The effect of faster transformations on average memory access time depends on the frequency of shared data accesses, as well as the magnitude of the transformation time relative to the other components of shared memory access time. The next chapter will develop performance models to address these issues. The results of those models are shown in the last column of Table 4-4.

Table 4-4: Shared data transformation and access time comparison

Implementation	Cycles per transform.	Cycle time (ns)	Transform. time (ns)	Access time (ns)
PAL	2	10	20	
TI Gate Array	1	<60	60	
Specialized instruction	1	60	60	822
Augmented instruction	35	60	2,100	2,784
Gen'l. purpose instruction	60	60	3,600	4,284
Nectar	–	–	–	30,000
Agora (read)	–	–	–	56,000

Chapter 5

Shared Memory Performance Models

The previous chapter considered alternative methods and locations for performing data type transformations. This chapter focuses on the design of a heterogeneous shared memory multiprocessor with a local hardware transformation unit.

The performance of a system using hardware transformation units is affected by the workload and the choice of at least seven design variables. Alternative designs resulting from these choices are compared using analytical performance models. The parameters in these models are assigned baseline values taken from values reported in the current literature. This allows relative comparisons of alternative designs. The values of some key parameters are then varied from the baseline to determine their effect on performance.

Results show that hardware transformation units, caching of unshared data, and local memory all provide significant performance advantages. Conversely, caching of shared data and the location of the transformation units have a less significant effect on performance.

5.1. Design Variables

Seven major variables affect the design of heterogeneous multiprocessors with hardware TUs. These variables, or design decisions, are shown in a taxonomy in Figures 5-1 and 5-2 and listed in Table 5-1. This research examines the branches shown in standard type (non-italicized) because these approaches reduce the complexity of the transformation and memory hardware. Each of these variables are described in detail in the following sections.

In Figure 5-1, the variables are grouped into three categories: Address Space Structure, Transformation Methods, and Cache. The Address Space Structure is affected by two factors: 1) the presence or absence of local memory and whether that local memory can be accessed by other processors, and 2) the amount of overlap between the address spaces of each processor. The branches under the Cache category represent three solutions to the cache coherency problem: do not use a cache, do not store shared memory (sm) data in the cache, or keep the shared data coherent.

The third category, Transformation Methods, is expanded in Figure 5-2. The data transformations can be performed either in software or hardware. A hardware Transformation Unit (TU) can be single-stage, pipelined, or integrated as part of a network. Each processor can have its own TU, or one or more central or distributed TUs could be shared by multiple processors. This research examines the use of hardware TUs since there has been little research in this area and this approach offers the potential for performance improvement. The models in Chapter 5 assume that each processor has its own local, single-stage TU. The TU can be located in the processor hardware, in a coprocessor, or in-line with the data bus. An in-line TU can be located either between the processor and the cache (P-TU-C-SM), or between the cache and the shared memory (P-C-TU-SM).

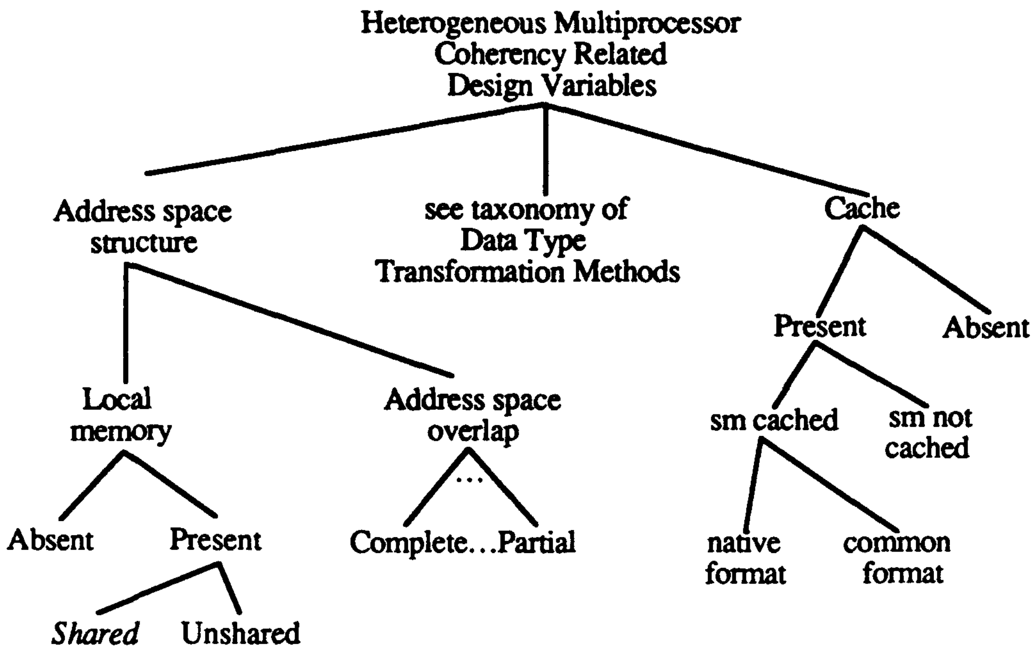


Figure 5-1: Taxonomy of coherency-related design variables

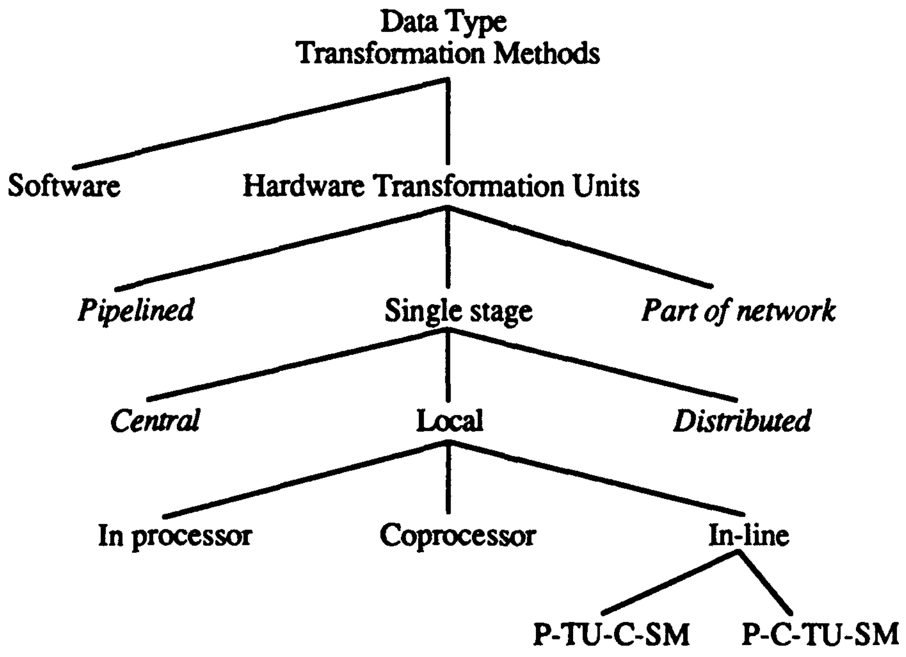


Figure 5-2: Taxonomy of data type transformation methods

Table 5-1: Coherency-related design variables

Category	Variables	
Address Space Structure	1. Address Space Overlap	Complete overlap
		Partial overlap
	2. Local memory present	
TU Location	3. In processor	
	4. Coprocessor	
	5. In-line	a. Between processor and cache
		b. Between cache and shared mem.
Cache	6. Cache present	
	7. Shared data cacheable	a. Common representation
		b. Native representation

5.1.1. Address Space (Memory) Structure

The address space, or memory, structure of a multiprocessor is affected by the amount of overlap between the address spaces of the constituent processors, and the presence of a local memory.

5.1.1.1. Address Space (Memory) Overlap

Many multiprocessors available today use a single, shared address space. Only one set of addresses exist, although all addresses may not be accessible to all processors. The address spaces of the different processors have complete overlap. In contrast, systems with multiple address spaces provide different address spaces for different processors, but they require some overlap to communicate between the address spaces. The memory addresses at this partial overlap are defined in both address spaces. This overlap is called shared memory and allows high speed communication between dissimilar processors. For a heterogeneous multiprocessor, partial overlap may be easier to implement than complete overlap because each address space does not have to be redefined (i.e. interrupt vector locations, operating system area, etc.). As shown in Figure 5-3, the amount of overlap can vary between complete overlap (i.e. a single address space) and partial overlap. Although the choice between complete and partial overlap may affect implementation, it does not significantly affect performance. As a result, this variable is not included in the evaluation of candidate designs in §5.3. Since a 16-bit processor such as the MIL-STD-1750A has a much smaller address space than a 32-bit processor, complete overlap is not possible. Part or all of the smaller address space can be overlapped with the larger address space (see §2.2. for a discussion of the address conversion).

This research assumes that data in shared memory is stored in a standardized, or common, representation (§3.3.4.). Data in non-shared memory is in the representation of the processor which uses that memory.

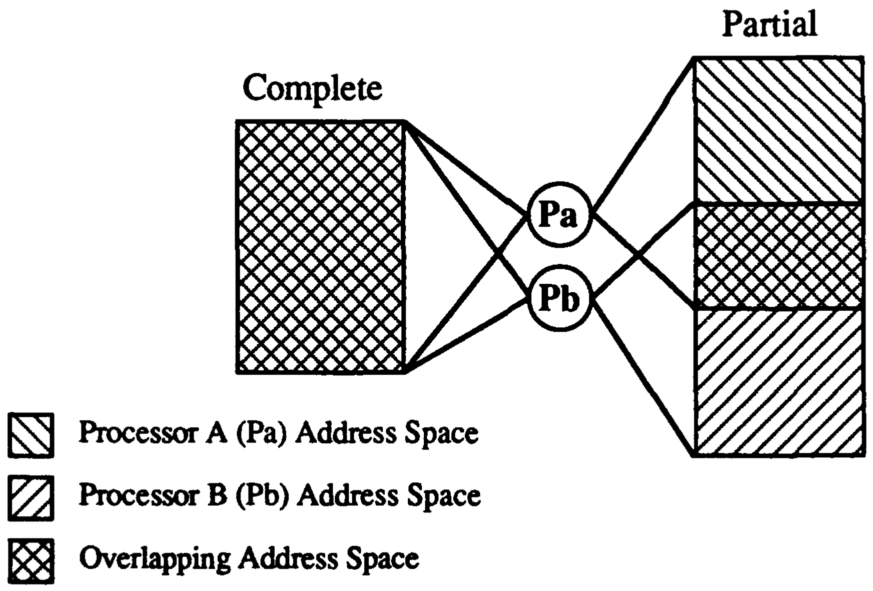

Partial
Complete
Pa
Pb
Processor A (Pa) Address Space
Processor B (Pb) Address Space
Overlapping Address Space

Figure 5-3: Address space overlap — complete vs. partial

5.1.1.2. Local Memory Presence

As an alternative to one large central memory, the main memory can be broken down into smaller units that are spread out among the processors. Shared data is kept in a smaller, shared, central memory while code and non-shared data are stored locally. In this way, only requests for shared data are delayed by contention for the interconnection network and shared memory. Local memory is in the processor's native representation and does not require transformations. This local memory is not a high-speed cache, but rather allows main memory to be distributed closer to the processors. This would be a block of addresses in a single address space design or a separate address space in a partial overlap design.

With a single address space, a local memory can either be accessible only by the local processor, or can be connected to the interconnection network and accessed by other processors. However, access to another processor's local memory makes memory coherence and transformations more complex than a shared memory in a common representation. This transformation complexity is discussed in [StCr88]. This research assumes that local memory can be accessed only by the local processor.

5.1.2. Transformation Unit (TU) Location

A hardware Transformation Unit (TU) can be located in the processor hardware, in-line with the data bus, or in a coprocessor.

5.1.2.1. In-processor TU

Transformation instructions implemented in hardware in the processor itself provide the highest performance of the three locations. Virtually no overhead is required to set up and read the TU, as with the other locations. However, this approach is only applicable to new processors. Existing processors must use either the in-line location or a coprocessor.

5.1.2.2. In-line TU

An in-line TU is located in-line with the data bus between the processor and the shared memory. If a cache is present, the in-line TU can be located either between the processor and the cache, or between the cache and shared memory. The location of the TU relative to the cache determines whether the shared data stored in the cache will be in the same common representation used for the shared memory (§3.3.4.) or in the processor's native representation. An in-line location does not require additional cycles for writing(reading) data to(from) TU registers, as with the coprocessor approach.

Common Format Cache

When an in-line TU is located between the processor and the cache, lines are moved between the cache and shared memory without transformation and are stored in the cache in the common representation. This simplifies the transformation process because only single words, as opposed to cache lines, must be transformed as they are accessed by the processor. For each shared memory reference, the transformation instruction is written to the TU instruction register. The transformation instruction is inserted in the code by the linker and post-processor (§6.). Since most memory accesses are to local memory and do not require transformation, the TU automatically reverts back to the null, or straight-through, transformation after each shared memory access. The null transformation is faster than the other transformations. Each shared data access requires a transformation, which delays the processor. In addition, all accesses to local memory are slightly delayed by the null transformation.

Native Format Cache

When an in-line TU is located between the cache and the shared memory, shared data is stored in the cache in the processor's native representation. A complete cache line is transformed into the native representation when it is read into the cache. If a word in a cache line is modified, that cache line must be transformed back to the common representation when it is written back to shared memory.

The update policy between the cache and shared memory affects the transformation process. With a write-through update policy, transformation instructions inserted in the code set up the TU for all shared data reads and writes. A transformation is performed on all writes to shared data since shared memory is updated immediately. On shared data reads, a transformation is only performed on cache misses. If the shared data is already in the cache, a transformation is not needed. Since cache misses can not be predicted, the TU must be set up for all shared data reads, even though a transformation may not be necessary.

With a write-back update policy, the transformation instructions inserted in the code can still be used to set up the TU for shared data reads. But data type tags are required in the cache since a dirty cache line could be written back to shared memory on any memory access. For the same reason, a cache coherency protocol that updates the cache would also require tags. This requires a more complex cache and a TU capable of reading the tags. A logical extension to a tagged cache is to also tag the shared memory. This would completely eliminate the need to send setup instructions to the TU.

An in-line location between the cache and shared memory has three advantages. First, as shared cache lines are moved into the cache, they are transformed into the processor's native representation. Future accesses to data in that line will not require a transformation. Second, local memory accesses are not delayed by the TU. And third, the TU can remain set to the previous transformation, which may be the same as the next one, rather than reverting back to the null transformation. However, this location has two disadvantages. First, with a write-back update policy the cache must use tags to identify the data type. Second, if invalidations occur frequently, the advantage of the native representation cache may be offset by the overhead of transforming words in the line which are invalidated before they are used.

5.1.2.1. Coprocessor TU

The final location for the TU is in a coprocessor. Coprocessors can be interfaced with processors in two primary ways: as a memory-mapped (or I/O-mapped) peripheral, or as a processor extension. A third approach, which is a variation of memory-mapping, uses the address bus to communicate the instruction [Glas90]. In all cases, the coprocessor is at an address rather than in-line with the data bus.

With a memory-mapped coprocessor, the instructions and operands are written to registers on the coprocessor by explicitly coding memory write instructions. Similarly, the result is read from an output register. The coprocessor instruction and operand registers are mapped to specific addresses in the processor's address space. Any processor can use a memory-mapped coprocessor (as long as they both use the same datatypes). And multiple coprocessors can be used, each at different addresses. However, memory-mapping requires additional memory cycles to write the coprocessor instruction, write each operand, and read the result(s).

When a coprocessor is connected as a "processor extension", this means that the processor has been designed to automatically recognize coprocessor instructions and write them to the coprocessor. The user does not need to code the read/write instructions for sending the information to and from the coprocessor, which reduces the overhead time somewhat. This more tightly integrated approach requires that the processor be designed to recognize coprocessor instructions. Some processors allow for multiple coprocessors by specifying the coprocessor number as part of the opcode [M688 87] [MIPS88], while the Intel 80x86 only allows one coprocessor to use the built-in interface [Glas90]. In the latter case, additional coprocessors must be memory-mapped.

The third approach, used by the Weitek Abacus 3167, uses a 64K address block instead of one address for each register. The data bus carries the data while the address bus tells the processor which instruction to perform. So instead of sending a separate instruction to the coprocessor, the instruction is determined by the address to which the data is sent. This reduces the number of memory cycles required by sending the instruction and one operand simultaneously.

Both the memory-mapped peripheral and the Weitek 3167 approaches can be readily used by a transformation coprocessor. Memory mapping is well-understood and easy to implement. The approach used by the Weitek 3167 is a novel method for reducing overhead, as long as the address space is sufficiently large to accommodate the block required. The processor extension approach could only be used if the processor provides user-definable coprocessor instructions.

The coprocessor performance models in this research use a standard memory-mapped approach. The overhead time consists of writing the instruction to the coprocessor (t_{tuc}), writing the data to be transformed (t_{tui}), and reading the transformed result (t_{tuo}). The first two components of this overhead have been reduced with the following two techniques. First, the previous transformation instruction is retained in the coprocessor instruction register, so a subsequent transformation instruction only needs to be sent if it is different from the previous instruction (p_{sd}). This is determined by the compiler post-processor (§6.2). Secondly, data only needs to be written to the coprocessor on shared memory *writes*. This is because on shared memory *reads*, the coprocessor reads the value from the data bus and performs the transformation automatically. The TU can be programmed to recognize only shared memory addresses, or it can simply transform everything that appears on the data bus since the processor will only read the TU for data that actually required transformation.

The coprocessor connection has two advantages: the TU does not delay the majority of the accesses which do not require transformation, and the instruction needs to be written to the TU only when a subsequent transformation is different from the previous one. The disadvantage is an extra cycle(s) is required to read the result from the TU output register.

5.2. Performance Model Parameters

This section describes the parameters that are used in the performance models and assigns them values. The performance model parameters are broken down into two categories: those that are primarily a function of the program, or workload (Table 5-2), and those that are primarily a function of the architecture, or implementation (Table 5-3). At this level of modeling, some of the parameters are affected by both the architecture and the workload. For example, the probability of a cache miss is affected by both the cache size, which is an architectural implementation issue, and the locality of the program, which is a function of the workload. For consistency, all the probabilities (p_{xx}) are listed under workload parameters, while times (t_{xx}) are listed under architecture parameters. The cache access time (t_c) is defined as the standard time unit and all other time values are stated as multiples (or fractions) of this parameter (the cache access time is assumed to be the same as the processor cycle time).

In order to establish a reference point in this multi-dimensional design space, this research uses one possible set of parameter values as a baseline to compare alternative designs. Although other values can be used, these appear to be reasonable starting points. Wherever possible, these parameter values have been taken from published studies. If published values were not available or were unknown, the assigned values are believed to be representative of existing architectures and workloads. A designer can change parameter values easily or vary them across a range to determine their effect on performance. Finally, since virtually no data for heterogeneous multiprocessors is available, values for homogeneous multiprocessors have been used.

5.2.1. Workload Parameters

Those performance model parameters which are primarily a function of the program, or workload, are listed in Table 5-2.

Table 5-2: Workload parameters and baseline values

Parameter	Description	Baseline Value
p_{cm}	probability of a cache miss	0.10
p_{cms}	probability of a cache miss for shared memory data	0.40
p_{dL}	probability of a dirty line in the cache	0.25
p_{sd}	probability a subsequent transformation is different	0.25
p_{sm}	probability of a shared memory reference	0.13
p_r	probability a shared reference is a read	0.92
p_w	probability a shared reference is a write	0.08

Studies done by Smith [Smit85], and Eggers and Katz [EgKa88] [EgKa89] provide values for many of these probabilities. Smith provides statistics for the relative frequencies of instruction fetches, data reads, and data writes based on 49 traces from 6 machine architectures. Based on traces from the IBM 370 and DEC VAX, he indicates that, as a rule of thumb, half of all memory references are data accesses, and that one-third of these are writes. Thus, the relative frequency of memory references are 50% instruction reads, 33% data reads, and 17% data writes (Figure 5-4).

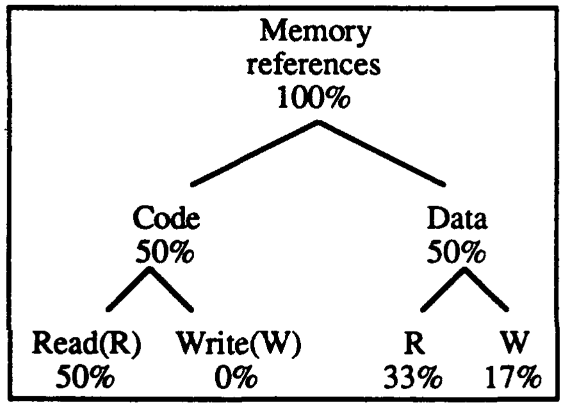


Figure 5-4: Relative frequency of memory references [Smit85]

Smith provides another rule of thumb that half of the *data* lines in a write-back (also called copy-back) cache will be dirty. Thus, assuming that a unified cache which holds both instructions and data is used, that code is not self-modifying, and that 50% of the cache lines are data, then 25% of these lines will be dirty (p_{dL} in Table 5-2). In the performance models, the term (1+pdL) accounts for the fact that a line is always read in on a cache miss, and is written back to the cache if the line to be replaced is dirty.

Eggers and Katz [EgKa88] provide relative frequencies based on four trace statistics from *parallel* CAD programs. One of the significant aspects of this work is their measurement of the frequency of shared data (p_{sm} in Table 5-2). Three of their traces are from a 12 processor Sequent machine running Unix and the fourth is from a 5 processor ELXSI 6400 running Embos. The mean values from the four traces are shown in Figure 5-5. Again, these frequencies for homogeneous multiprocessors are used in the following performance models since data is not available for heterogeneous multiprocessors. Because p_{sm} is so workload dependent and is a key parameter in the performance of the memory system, it will be the subject of further discussion and sensitivity analysis in §5.3.3. (Figure 5-18).

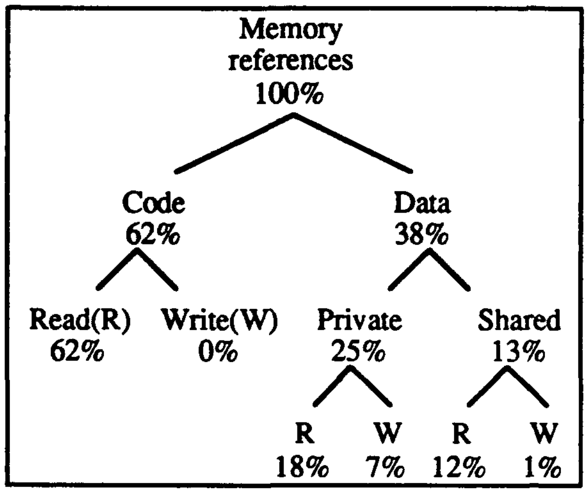


Figure 5-5: Relative frequency of memory references [EgKa88]

The parameter p_{sd} accounts for the fact that with a coprocessor TU, and an in-line TU between the cache and shared memory, an instruction needs to be written to the TU only if it is different from the previous one.

5.2.2. Architecture (Implementation) Parameters

Those performance model parameters which are primarily a function of the architecture are listed in Table 5-3 (to clarify the choice of parameter names, the letters used in the parameter name are capitalized in the description column).

Table 5-3: Architectural parameters and baseline values

Parameter	Description	Baseline Value
L	cache Line size in 32-bit words (not bytes)	4
N	Number of processors	5
t_c	Cache access Time (defined as one time unit) :=	1
t_{cc}	Cache Coherency Time	0.5
t_m	main/shared Memory access Time	4
t_{m2}	main/shared Memory sequential word access Time	1
t_{mL}	main/shared Memory Line access Time	7
t_{ma}	Average Memory access Time	Calc
t_{net}	interconnection NETwork latency	2.0
$t_{q\,x}$	Queue (wait) Time for the IN and shared memory	Calc
t_{sw}	average transformation Time in SoftWare	35
t_{tuc}	write Time for the TU Control (instruction) register	1
t_{tui}	write Time for TU Input register	1
t_{tun}	TU Null transformation Time	0.3
t_{tuo}	read Time for TU Output register	1
t_{tut}	TU average Transformation Time	1

An embedded (§1.) heterogeneous multiprocessor is expected to have a relatively small number of processors (i.e. N=5 above). The performance advantage of such a system is a result of the specialized processors rather than from massive parallelism. The cache coherency time reflects the degradation on performance caused by maintaining coherency (see next section). Although both shared memory and local memory have the same chip access time (t_m), access to shared memory is delayed by the latency of the interconnection network (t_{net}) and the queueing time (t_q). The average transformation time in software (t_{sw}) includes the time for switching context, reading in the instructions and data, performing the transformation and returning to the previous context (§4.2.2.).

5.2.2.1. Cache Assumptions

This section explains the assumptions made regarding the caches. A single set of assumptions is a simplification since each processor and cache controller may work in a slightly different manner. However, these assumptions allow accurate first order comparisons of alternative heterogeneous multiprocessor designs.

The cache is a small high speed memory which is faster than either the local memory or shared memory. As discussed previously, it is more difficult to maintain cache coherency on heterogeneous multiprocessors due to different data types and representations. Three possible solutions are: do not use a cache, do not store shared data in the cache, or store both shared and unshared data in the cache and keep the caches coherent. All of these approaches are evaluated.

These models assume a unified cache containing both instructions and data. A unified cache, rather than a split cache, is frequently required for upgrading systems because old programs, which still must be run, were written with self-modifying code [Craw90]. The unit of transfer between cache and main memory is one line (sometimes called a block). The processor sends all memory requests to the cache first and, if a miss occurs, that line and word is brought into the cache and processor, respectively, from the local or shared memory. If a processor writes to a line in the cache, it is called a dirty line and the entire line is written back to the memory either immediately (write-through) or when the line is replaced (write-back). All but one of the designs in this research use write-back. Design J, which has a native representation cache, uses write-through to avoid the need for tags in the cache and shared memory.

The benefit of caching shared data is somewhat offset by the cost, in both hardware and performance, of maintaining coherency. Baylor and Briggs [BaBr89] report that a cross-interrogate protocol degrades the overall performance of parallel PDE algorithms by 10 to 30 percent. This research does not model specific cache coherency protocols, since the intent of this research is to compare alternative system designs.

The effect of coherency protocols on performance is modeled primarily through the values assigned to p_{cms} (probability of a cache miss for shared memory data) and t_{cc} (cache coherency time). The probability of a cache miss for shared memory data (p_{cms}) is greater than the probability of a cache miss for native memory data (p_{cm}). This is due to the cache invalidation which occurs when another processor writes any variable in that cache line. For the baseline, $p_{cms} = 4*p_{cm}$ to reflect invalidation of shared data. Results from [EgKa89] indicate that the probability of a miss for shared data is approximately nine times greater than unshared data. However, this data is based on steady state conditions (cache already filled – "warm start") on homogeneous multiprocessors. A cold start analysis, which is more appropriate for analytical performance models, would increase the miss ratio for unshared data more than shared data. This is because a very high percentage of the shared data misses are invalidation misses.

The cache coherency time is added in on every shared memory access and it reflects the fact that cache coherency operations delay the processor from accessing the cache.

5.2.3. Queueing Model for Shared Memory

Shared memory accesses are delayed when there is contention with other processors for shared memory. This delay is modeled with a closed, finite population queueing model [Ferr78]. The use of a queueing model assumes that a queue exists in some form (either real or virtual) to control access to a single-user interconnection network and shared memory. The closed, finite population model is based on the assumption that a processor cannot submit a new shared memory request until the previous request has been satisfied.

The three inputs to the queueing model are the number of processors, N; the average time between shared memory accesses, Et; and the service time, Es. The time between shared memory accesses is different for each design and is a function of the probability of a shared memory reference, the presence of a local memory and/or cache, and the cacheability of shared memory data. The service time also varies for each design and is of the form

$$Es = t_{net} + p_1 * t_m + p_2 * t_{mL}$$

where p_1 and p_2 are the relative frequencies of single-word and cache-line accesses to shared memory. Single-word accesses to shared memory occur in designs that do not permit caching of shared data. The formulas and values for Et and Es, as well as the other factors in the queueing models, are given in Appendix B.

The output of the queueing model is the mean response time, W, which is calculated by the formula

$$W = \frac{N * Es}{1 - p_0} - Et$$

The term p_0 is the probability that shared memory is idle, and it is calculated by the formula [Klei76] [Alle80]

$$p_0 = \left[\sum_{n=0}^{N} \frac{N!}{(N-n!)} \left(\frac{Es}{Et}\right)^n \right]^{-1}$$

By definition, the mean response time, W, is the sum of the service time plus the queueing time:

$$W = Es + t_q$$

The queueing time is calculated by simply subtracting the service time Es from the value calculated for W:

$$t_q = W - Es$$

5.3. Example Designs and Performance Models

This section uses analytical performance models to examine the effects of the design variables on ten candidate designs (Table 5-4). Analytical performance models provide first-order estimates of the relative performance of alternative designs. These models allow alternative topologies to be compared early in the design process. One advantage of these analytical performance models is that they allow a designer to change parameter values easily or vary them across a range to determine their effect on performance. The candidate designs provide a cross-section of the possible combinations of the design variables.

First, a coprocessor TU is used to compare the impact of the six combinations of local memory, caching of unshared data, and caching of shared data (Designs A - F, Table 5-4). The coprocessor TU is then compared to an in-processor TU, to an in-line (IL) TU at two locations, and to software transformations. These last five designs (Designs F - K) are all identically configured with local memory, a cache, and caching of shared data.

Table 5-4: Design variable combinations for candidate designs

Variables	Candidate Designs									
	A	B	C	D	E	F	G	H	J	K
Local memory		√		√		√	√	√	√	√
Cache-unshared			√	√	√	√	√	√	√	√
SM cacheable					√	√	√	√	√	√
Coprocessor	√	√	√	√	√	√				
In processor							√			
IL P-TU-C-SM								√		
IL P-C-TU-SM									√	
Software										√

Figures 5-6 through 5-13 show the topologies for these architectures. For drawing simplicity, these figures all show two processors (Pa and Pb) on a bus, but actual designs can have additional processors and use other types of interconnection networks (IN). The performance metric for comparing these designs is the average memory access time, t_{ma}. This includes the time required to transform shared variables into the processor's native representation. Average memory access time directly affects processor performance, as long as it is slower than the processor cycle time [Gill87]. The specific performance model for each design is listed in Table 5-5.

Design A has coprocessor TUs (TUa and TUb) with no local memory or cache. The single shared memory is shown divided into sectors. Shared data is stored in the shared memory (SM) sector in the common representation. Unshared data such as code is stored in the corresponding processor's sector (Ma and Mb) in that processor's representation and does not require transformation. Design B has coprocessor TUs and local memory (LMa and LMb). Design C has coprocessor TUs with caches (Ca and Cb) for unshared data only. Design D has coprocessor TUs, local memory, and unshared cache. Design E has coprocessor TUs with a cache for both shared and unshared data. Design F has coprocessor TUs, local memory, and a cache for both shared and unshared data.

Designs F through K all have local memory and a cache for both shared and unshared data, but they differ in the location of the TU: F has a coprocessor, G has the TU integrated as part of the processor hardware, H has the TU in-line between the processor and the cache, J has the TU in-line between the cache and shared memory, and finally, K performs the transformations in software.

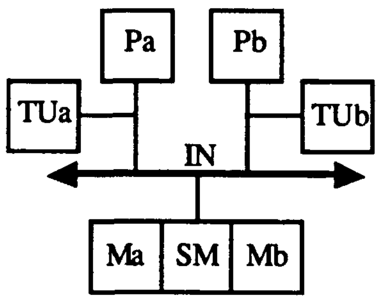


Figure 5-6: Design A

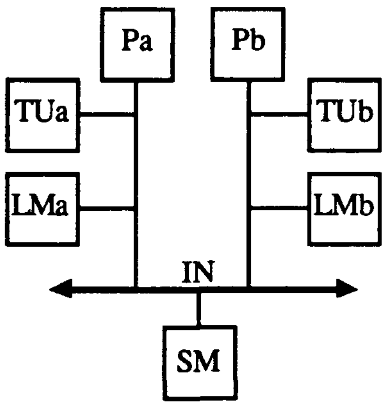


Figure 5-7: Design B

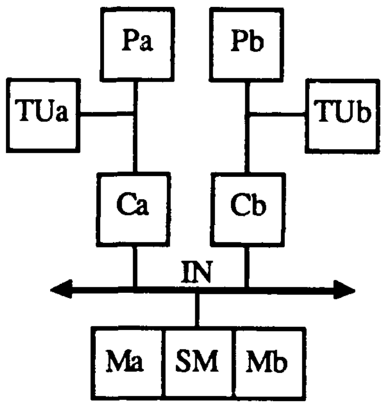


Figure 5-8: Design C (unshared cache), Design E

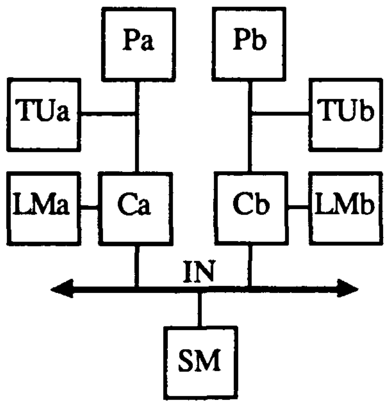


Figure 5-9: Design D (unshared cache), Design F

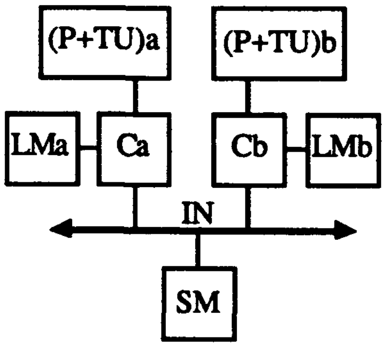


Figure 5-10: Design G

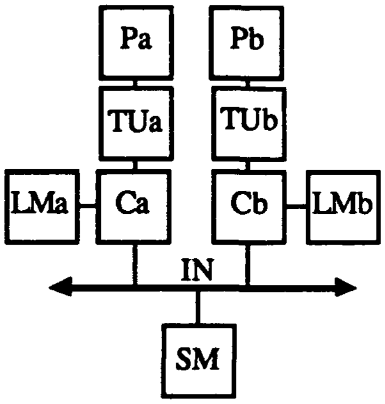


Figure 5-11: Design H

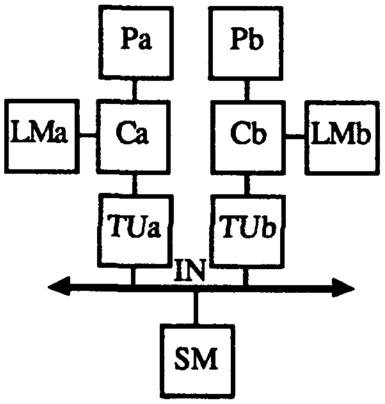


Figure 5-12: Design J

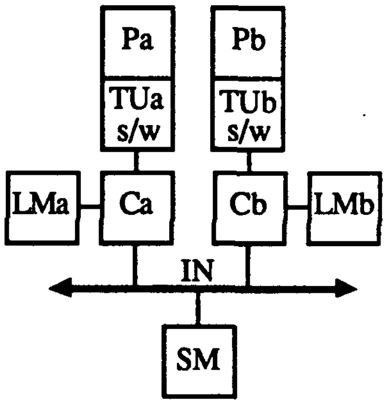


Figure 5-13: Design K

Table 5-5: Specific performance models

Design	Performance Model
A	=psm*(psd*ttuc+pw*ttui+ttuo+ttut+(tqa+tnet+tm))+ (1-psm)*(tqa+tnet+tm)
B	=psm*(psd*ttuc+pw*ttui+ttuo+ttut+(tqb+tnet+tm))+ (1-psm)*tm
C	=psm*(psd*ttuc+pw*ttui+ttuo+ttut+tqc+tnet+tm)+ (1-psm)*(tc+pcm*(tqc+(1+pdL)*(tnet+tmL)))
D	=psm*(psd*ttuc+pw*ttui+ttuo+ttut+tqd+tnet+tm)+ (1-psm)*(tc+pcm*(1+pdL)*tmL)
E	=psm*(psd*ttuc+pw*ttui+ttuo+tc+tcc+ttut+ pcms*(tqe+(1+pdL)*(tnet+tmL)))+ (1-psm)*(tc+pcm*(tqe+(1+pdL)*(tnet+tmL)))
F	=psm*(psd*ttuc+pw*ttui+ttuo+tc+tcc+ttut+ pcms*(tqf+(1+pdL)*(tnet+tmL)))+ (1-psm)*(tc+pcm*(1+pdL)*tmL)
G	=psm*(tc+tcc+ttut+pcms*(tqg+(1+pdL)*(tnet+tmL)))+(1-psm)*(tc+pcm*(1+pdL)*tmL)
H	=psm*(ttuc+tc+tcc+ttut+pcms*(tqh+(1+pdL)*(tnet+t mL)))+(1-psm)*(tc+ttun+pcm*(1+pdL)*tmL)
J	=psm*(psd*ttuc+tc+tcc+pw*(tqj+tnet+tmL+L*ttut)+ *pcms*(tqj+(1+pdL)*(tnet+tmL+L*ttut)))+* (1-psm)*(tc+pcm*(1+pdL)*tmL)
K	=psm*(tc+tcc+tsw+pcms*(tqk+(1+pdL)*(tnet+tmL)))+(1-psm)*(tc+pcm*(1+pdL)*tmL)

The relationships between the performance model parameters are illustrated in a general taxonomy in Figure 5-6. Multiplicative factors are shown on vertical lines and additive factors on horizontal lines. All of the specific performance models can be derived from this taxonomy. The equation for the taxonomy and rules for deriving specific models are shown in Table 5-5. If the condition in a rule is not true, the parameters use their normal value and the non-applicable "switches" (C, CSM, LM, CP, ILc, and ILn) are set to zero.

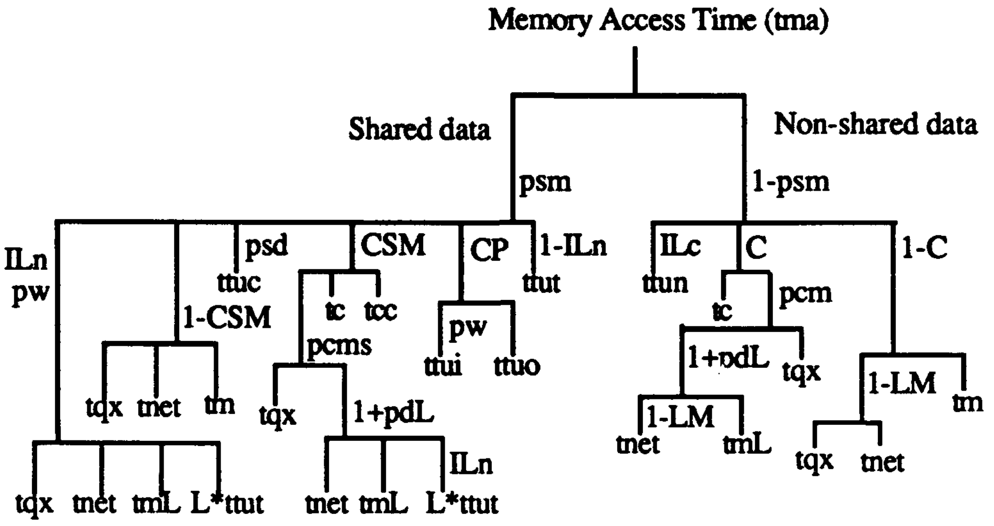


Figure 5-14: Taxonomy of performance model parameters

Table 5-6: Equation and rules for general performance model

tma = psm*(ILn*pw*(tqx+tnet+tmL+L*ttut)+ (1-CSM)*(tqx+tnet+tm)+psd*ttuc+ CSM*(tc+tcc+pcms*(tqx+(1+pdL)*(tnet+tmL+ILn*L*ttut)))+ CP*(pw*ttui+ttuo)+(1-ILn)*ttut)+ (1-psm)*(ILc*ttun+C*(tc+ pcm*(tqx+(1+pdL)* ((1-LM)*tnet+tmL)))+(1-C)*(tm+(1-LM)*(tqx+tnet)))
If cache, C=1
If cacheable shared memory, CSM=1
If in software, ttuc=ttun=0, ttut=tsw
If homogeneous, ttuc=ttut=0
If in-line P-TU-C-SM, ILc=psd=1
If in-line P-C-TU-SM, ILn=1
If local memory, LM=1
If coprocessor, CP=1
If in-processor, ttuc=0

5.3.1. Baseline Results

The average memory access time is computed for each design using the baseline values for the parameters as listed in Table 5-2 and 5-3. The results for these baseline values are shown in Figure 5-15 and Table 5-7. (Design A is usually significantly slower than the other designs. In some of the bar graphs that follow, the bar for Design A has been truncated and the scale expanded so that the differences between the other designs can be seen more easily. In these cases, the bar has been shaded to remind the reader that it has been truncated. The number at the top of the truncated bar represents the correct value.) Some of the design variables clearly have a more significant impact on performance than others. Local memory or unshared cache by themselves both provide significant performance improvements over a single shared memory (B vs. A: 80%, C vs. A: 71%). Local memory is 31% faster than an unshared cache (B vs. C). The combination of local memory and unshared cache together provides further significant improvement (D vs. B: 32%). However, caching of shared data provides a rather small improvement (E vs. C: 4%, F vs. D: 13%), which may not warrant the additional cost of cache coherency hardware. A hardware TU provides a significant improvement over software (F-J vs. K: 55%), although the location of the TU does not make a large difference (F vs. G-J: +5% to -11%). Design G, which has the TU integrated with the processor, has the highest performance although it is only slightly faster than a coprocessor (G vs. F: 5%).

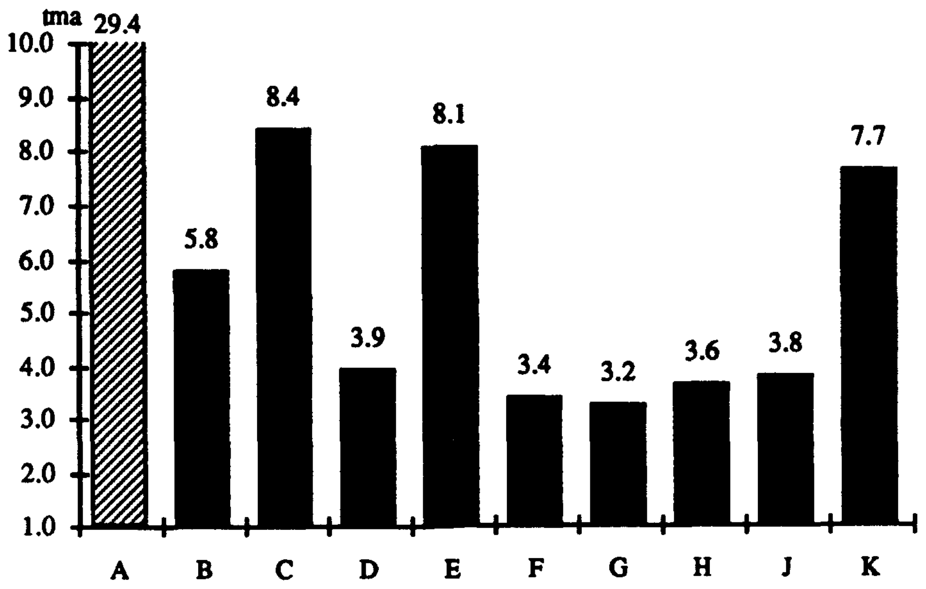


Figure 5-15: Baseline performance comparison (t_{ma})

Table 5-7: Baseline performance comparison (t_{ma})

Variation	Candidate Designs									
	A	B	C	D	E	F	G	H	J	K
Baseline	29.4	5.8	8.4	3.9	8.1	3.4	3.2	3.6	3.8	7.7
%Improvement		80	71	32	4	13	5	-6	-11	-125
relative to		A	A	B	C	D	F	F	F	F

In order to compare the shared memory access time of this approach to other approaches (§4.2.3), the average memory access time shown in Figure 5-15 can be broken down into shared data access time and unshared data access time. For Design F and Design K, these component times are shown in Table 5-8. Although the hardware TU is 35 times faster than the software transformation (Table 5-3), the transformation time is a relatively small part of average memory access time. As a result, the average *shared data* access time for Design F is only 3.4 times faster than Design K (Table 5-8 – 46.4÷13.7) because of the other times shown in the performance models such as queueing, cache coherency, cache miss, etc. Furthermore, since they both have the same access time for unshared data, the average memory access time for Design F is only 2.26 times faster than Design K (Figure 5-15). Nevertheless, the ability to cut average memory access time in half is still quite significant.

Table 5-8: Components of memory access time for Design F and Design K

Component	Frequency (p_{sm})	Design F Times (t_c)	Design K Times (t_c)
Shared data	0.13	13.7	46.4
Unshared data	0.87	1.9	1.9

5.3.2. Variations from Baseline

The values for some of the key parameters can be varied from the baseline to determine the effect on the performance of the ten designs. The baseline value for the interconnection network latency time (t_{net}=2) is representative of a bus. One possible variation on this is a multistage interconnection network (MIN), in which the latency varies with the number of stages and is proportional to $\log_2 N$, where N is the number of source nodes and the in-degree=2 [LiMa87]. The results for a *non-blocking* network with t_{net}=8 are shown in Figure 5-16 and Table 5-9. The effect of using a non-blocking network is that there is no queueing for the network, only for the shared memory. The queue now forms between the network and shared memory instead of between the processors and network. This is modeled by removing t_{net} from the service time (§5.2.3.). Even though the network latency is greater, the reduction in service time is enough to offset that additional latency for Designs A, C, and E. For Designs A, C, and E the shared memory is saturated (Table B-1, p_0=0), so any reduction in service time has a good potential for decreasing average memory access time.

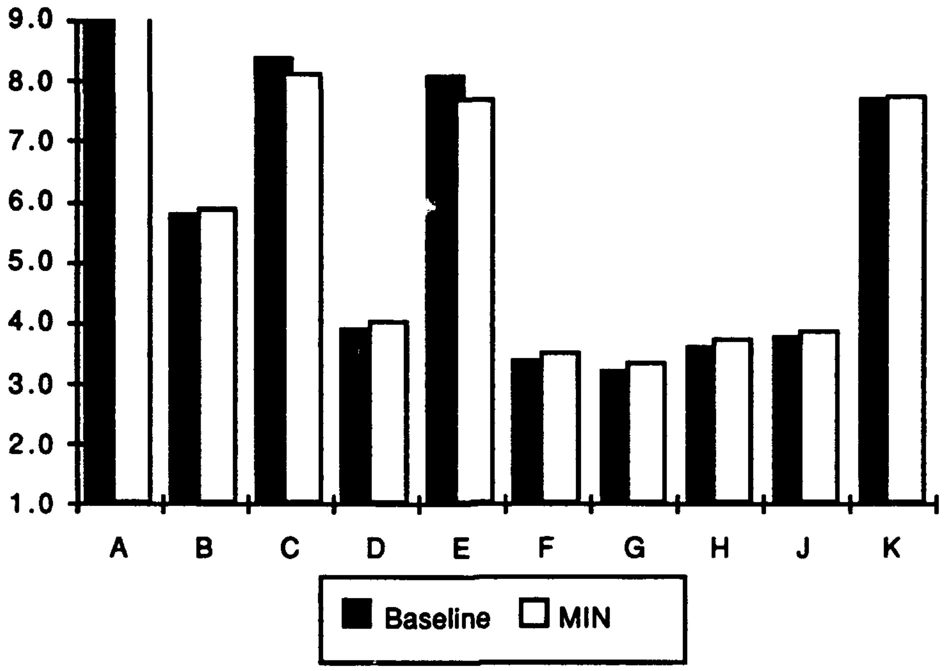


Figure 5-16: Performance (t_{ma}) of a non-blocking multistage interconnection network (t_{net}=8) relative to the baseline (Design A off scale, see Table 5-9)

Table 5-9: Performance (t_{ma}) of a non-blocking multistage interconnection network (t_{net}=8) or a slower transformation unit (t_{tut}=4)

Variation	Candidate Designs									
	A	B	C	D	E	F	G	H	J	K
Baseline	29.4	5.8	8.4	3.9	8.1	3.4	3.2	3.6	3.8	7.7
MIN	27.4	5.9	8.1	4.0	7.7	3.5	3.3	3.7	3.9	7.8
Slow TU	29.8	6.2	8.8	4.3	8.5	3.8	3.6	4.0	4.7	7.7

Similarly, the impact of a slower TU can be determined by increasing the transformation time by a factor of four (t_{tut}=4). This change does not have a significant effect on overall performance, which indicates that the performance advantage of a hardware TU is relatively insensitive to its speed (Table 5-9 and Figure 5-17). The slower TU still provides around a 45% improvement over software.

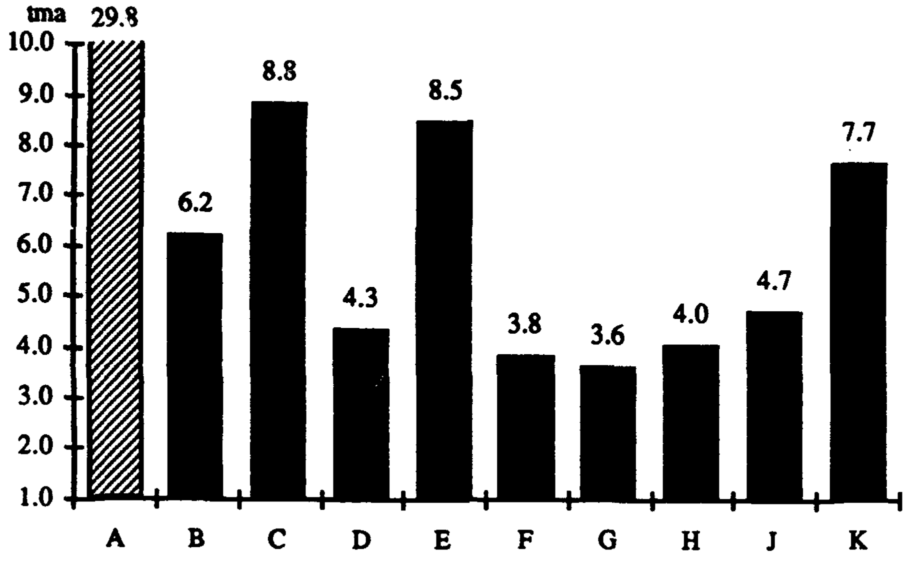


Figure 5-17: Performance (t_{ma}) of slow Transformation Unit (t_{tut}=4)

The final parameter considered here is the probability of shared memory references (p_{sm}). The performance for $0 \leq p_{sm} \leq 14\%$ is shown in Figure 5-18 (Design A is off this scale - it is a nearly horizontal line at 29.2). The benefit of a hardware transformation unit is proportional to the frequency of shared memory accesses. If, for example, only one or two percent of the memory references are to shared memory, hardware TUs do not provide a significant performance improvement over software transformations (Design K). The Y-intercept, which corresponds with p_{sm}=0, is the access time for non-shared memory. The difference in the Y-intercept between Designs B and C/E reflects the advantage of a cache over local memory. The difference between Designs C/E and F/G/J/K reflects the advantage of adding local memory to a cache. The difference between Design H and F/G/J/K reflects the null transformation time (t_{tun}) for the in-line TU. The slope of the lines approximates the cost of a shared memory access. The steepness of Design K reflects the cost of doing the transformations in software.

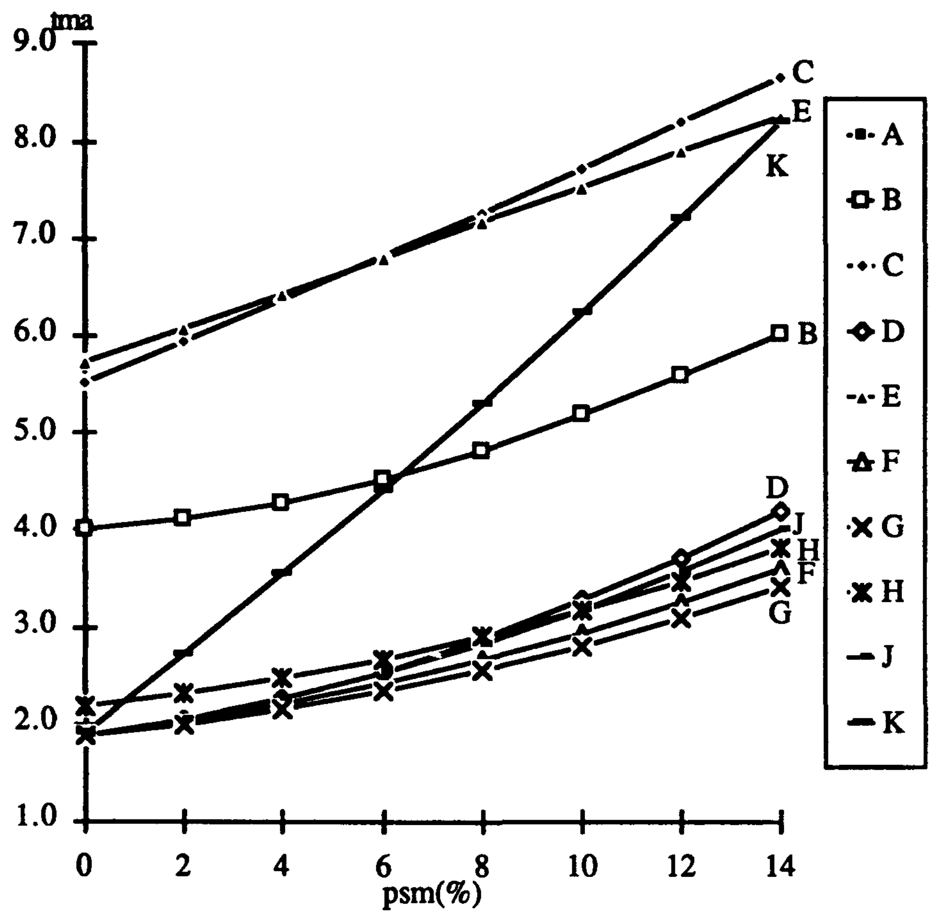


Figure 5-18: Performance for $0 \leq p_{sm} \leq 14\%$ (A off scale at 29.2, F between G and H)

The curve in the lines in Figure 5-18 is caused by the increase in queueing time as p_{sm} increases. As p_{sm} approaches 1.0, queueing time asymptotically approaches the value [(N-1)*Es-Et] (see Figure 5-19). If all memory references are to shared memory, each processor must get back in the queue and wait while the other N-1 processors are serviced [(N-1)*Es]. That time is reduced by the time between requests, Et, since the processor does some processing before its next shared memory request.

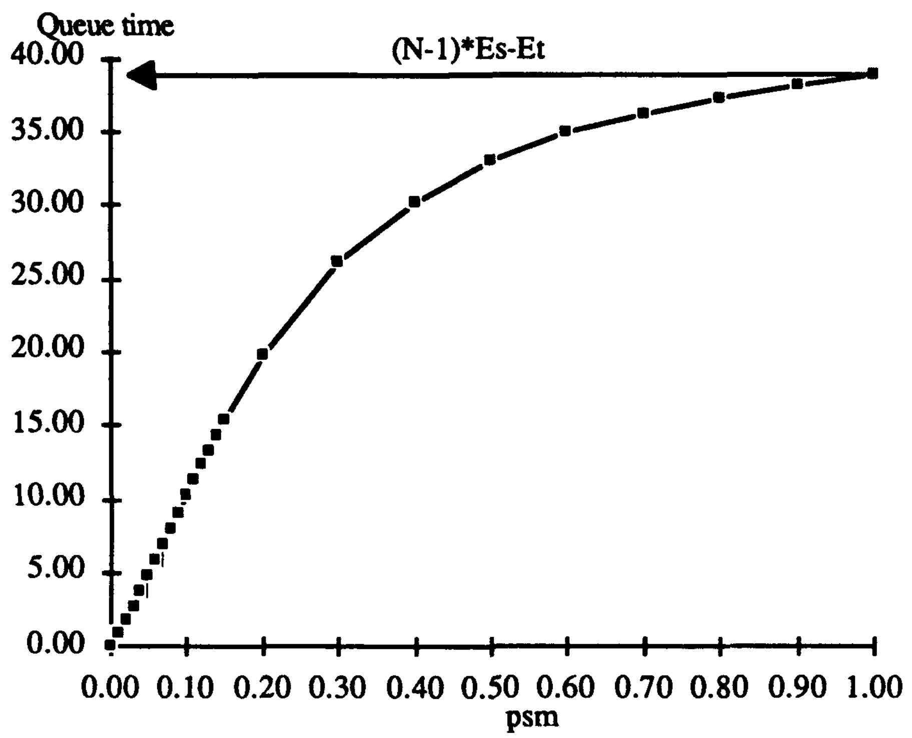


Figure 5-19: Queueing time (t_q) as a function of p_{sm} (Design F)

5.4. Conclusions on Performance Models

Analytical performance models allow a designer to quickly estimate the relative performance of different designs. Since exact parameter values can be difficult to predict, they can easily be varied to determine the sensitivity of a design to changes in a particular parameter. The models presented in this chapter show that significant performance advantages are provided by hardware transformation units, caching of unshared data, and local memory. Caching of shared data and the location of the transformation units have a less significant effect on performance.

For a situation similar to the baseline defined in this chapter, Design D, which has a coprocessor TU, local memory and unshared cache, provides good performance with very little additional hardware. The unshared cache avoids the cache coherency problem, while the coprocessor TU can be easily added to an existing system design. Although the in-processor TU in Design G provides the highest performance, it may not be the most efficient use of silicon area for a general purpose microprocessor which only has a small percentage of its applications in heterogeneous systems.

Chapter 6

Required System Software

In order to use hardware transformation units in an untagged heterogeneous multiprocessor, additional system software is needed at compile time. This system software must (1) communicate the addresses and data types of shared data between processors and (2) insert instructions for the TU into the compiled code. These two functions can be accomplished with a heterogeneous linker and a post-processor for the compiled code.

6.1. Heterogeneous Linker

In a heterogeneous system, the compiler for each processor must create a symbol table containing the following information for shared data:

-processor type and ID

-shared variable name

-data type

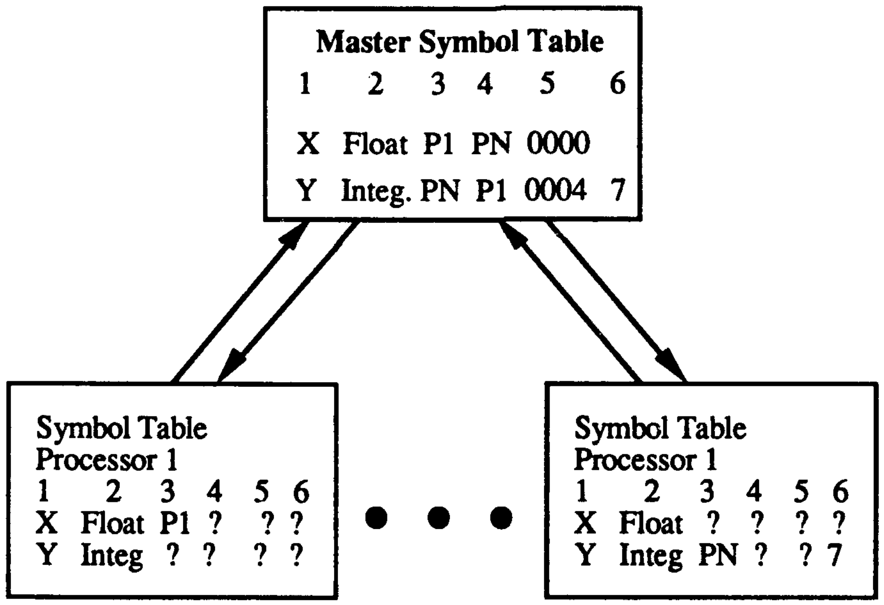
-source processor

-destination processor(s)

-value

-address

The compiler must provide the first three of these items; some of the remaining items may not be known and will be added to the table by the heterogeneous linker. The linker performs this function by 1) gathering all information on shared data from the individual symbol tables and then 2) combining this information in a master symbol table (Figure 6-1). The linker will normally assign shared data an address in shared memory. Since the linker knows the source and destination data types, it also specifies the desired transformation. If the linker observes that a particular item of data is shared only by identical processor types, it can specify a null transformation and store the data in that processor's native representation. The completed master symbol table is then used to fill in the unknown information in each of the individual symbol tables. Finally, the completed individual symbol tables are sent back to the individual compilers where they are used by the post-processor.



1. Shared Variable Name
2. Data type
3. Source processor
4. Destination processor
5. Address
6. Value

Figure 6-1: Symbol table information used by heterogeneous linker

6.2. Compiler Post-processor

Upon the return of the completed symbol table from the heterogeneous linker, a post-processor inserts the appropriate instruction for the TU at the proper place in the code. Alternately, the compiler could be modified to perform this function. A smart compiler could optimize the location of the setup command based on other activities of the processor. If the TU is a coprocessor (Designs A-F) or is located between the cache and shared memory (Design J), a new instruction is required only if the type of transformation changes (p_{sd}). On the other hand, if the TU is in-line between the processor and the cache (Design H), it must revert to the null transformation after each shared memory access so that local memory accesses will not be transformed. As a result, the in-line location requires a new instruction for each shared memory access.

Chapter 7

Conclusion

Heterogeneous shared memory computer systems pose additional challenges not found in homogeneous systems. This research has considered the design issues involved in sharing primitive data types and has proposed designs which provide high-speed sharing of data.

As with any cost-performance tradeoff in computer design, the value of specialized hardware varies with the frequency of its use. In this case, the value of a hardware TU varies with the probability of shared memory accesses. At low probabilities, the resulting performance advantage of the hardware TU would be small and cost considerations would suggest that the transformations be done in software. However, at higher probabilities, a hardware TU can cut the average memory access time in half (Figure 5-15).

The primary applications for these type of designs are in special purpose applications which require maximum performance and tight coupling between heterogeneous processors. The linking that must be done at compile time makes these designs less suited for general purpose applications and development work.

The analytical performance models presented in this research demonstrate the performance advantages of a shared memory heterogeneous multiprocessor. A coprocessor-type TU (Design F) is 2.26 times faster than a software implementation and can use standard memory technology for the shared memory (i.e. a tagged memory is not required). These performance models allow a designer to quickly estimate the effect of design changes on performance.

The original proposal for this research considered the use of coprocessor-type transformation units for existing processors. In addition, this research shows that new processors can improve the performance of a heterogeneous multiprocessor by augmenting their instruction sets to include transformations. However, the effective performance of a processor with an augmented instruction set (Design G) is only slightly faster (+5%) than the coprocessor TU (Design F).

It is difficult to verify the performance of a heterogeneous shared memory multiprocessor with hardware TUs since none have been built. As with the original cache papers [Wilk65], verification must await the development of prototypes. Recent developments such as processor byte reordering and the TI gate array indicate that some of the necessary transformations are being implemented in hardware. Similarly, Motorola's recent system design, which combines a general purpose RISC processor with DSP processors, demonstrates a growing awareness of the benefits of heterogeneous multiprocessors.

The use of heterogeneous multiprocessors will continue to grow as designers combine specialized processors to provide high performance solutions to specific problems. The need for transformations will depend on the mix of processors. Designs which use incompatible processors must provide transformations in either hardware or software. This research provides answers to many of the problems that arise when sharing memory between heterogeneous processors.

Appendix A

Floating Point Transformation (LISP)

```
;;; -*- Mode:Common-Lisp; Fonts:(MEDFNT); Base:10 -*-

;;; This LISP program transforms an IEEE Floating Point number to a two's
;;;  complement floating point number.  It also contains extensive test vectors to
;;; demonstrate the correctness of the transformations for the special cases.

;;; NOTE on bit-vector data structure:  Lisp treats all bit-vectors as being
;;; 32 bits long. So, for example, the ieee-exp is actually 32 bits long
;;; even though we only use the lower 8 bits.  When we first create ieee-exp,
;;; GET-BITS  sets the upper 24 bits to zero.  And when we print it, we only
;;; print out the lower 8 bits.     One problem with printing out these
;;; bit-vectors is that if Lisp sees a one in bit 31, Lisp thinks the
;;; bit-vector is a negative number and prints it as such.  The way around
;;; that problem is to AND the bit-vector with a string of ones before
;;; printing it:  (logand #xffffffff bit-vector).  This somehow prevents Lisp
;;; from interpreting the bit-vector as a two's complement number.
;;; This only needs to be done when bit 31 could be a one.


;;; GET-BITS: Extracts the bits from bit-vector between start-bit and end-bit,
;;; inclusive.  Returns a 32 bit bit-vector in which the lsb is the start-bit
;;; in the original bit-vector.  The upper bits are set to zero.
;;; I wrote this function so that, for example, you can get-bits 0 to 22, or
;;; 22 to 0.  The arguments can be in either order.  They both return the same
;;; string of bits.

(defun get-bits(start-bit end-bit bit-vector)
  (ldb (byte (+ (abs (- end-bit start-bit)) 1) (min start-bit end-bit))
      bit-vector))

;;; GET-BIT is similar to GET-BITS except it just gets one bit.
;;; This bit is returned as the lsb.
;;; The remaining 31 bits are set to zero
;;; EX: (get-bit 22 tc-bit-vector) ;gets bit 22

(defun get-bit (bit-pos bit-vector)
  (ldb (byte 1 bit-pos) bit-vector))


;;; PUT-BITS: Deposits new-bits into bit-vector between start-bit and end-bit,
;;; inclusive.  Returns a copy of the modified bit-vector.  Ignores any extra
;;; bits at the upper end of new-bits.
;;; Another approach is to write it with a setq so it actually
;;;   modifies bit-vector.
;;; EX: (setq tc-bit-vector (put-bits tc-exp 24 31 tc-bit-vector))

(defun put-bits(new-bits start-bit end-bit bit-vector)
  (dpb new-bits (byte (+ (- end-bit start-bit) 1) start-bit) bit-vector))


;;; SET-BIT is similar to PUT-BITS except it just sets one bit to 1 or 0.
;;; EX: (set-bit 22 1 tc-bit-vector)  ;sets bit 22 to 1

(defun set-bit (bit-pos value bit-vector)
  (dpb value (byte 1 bit-pos) bit-vector))

;;; LEFT-SHIFT shifts a bit-vector left by number-of-bits
;;; It's a little bit more mneumonic than lsh, and the number-of-bits is the
;;;  first argument which is easier to read.

(defun left-shift (number-of-bits bit-vector)
  (ash bit-vector number-of-bits)
)

;;; RIGHT-SHIFT shifts a bit-vector right by number-of-bits
;;; It's a lot more clear than lsh by a negative amount, and the number-of-bits
;;;  is the first argument which is easier to read.

(defun right-shift (number-of-bits bit-vector)
  (ash bit-vector (- number-of-bits))
)

;;; COMPLEMENT-BITS is used for complementing the fraction portion of a
;;; bit-vector.  Returns a copy of the bit-vector with the fraction bits
;;; complemented.
;;; Ex: (setq tc-bit-vector (complement-bits 0 23 tc-bit-vector))

(defun complement-bits (start-bit end-bit bit-vector)
  (put-bits
   (+ 1 (lognot (get-bits start-bit end-bit bit-vector)))
   start-bit end-bit  bit-vector)
)

;;; IEEE-TO-TC is the function to call on an ieee bit-vector to make a
;;; two's complement bit-vector.  The hardware transformation unit would perform
;;;  this function.  It assumes a hidden bit and a +127-biased exponent.
;;; Exceptions (special cases) are corrected later by check-flags.

(defun ieee-to-tc(bv)
  ; Declares don't prevent lisp from interpreting leading ones as meaning a
  ; negative number.  I'm leaving this here (preceded by a semi-colon to make
; it a comment) to remind me it doesn't help any.

;(declare (type (unsigned-byte 32) bv))

;declare all these variables special to avoid compiler warnings.
(declare (special ieee-frac))
(declare (special ieee-exp))
(declare (special ieee-sign))
(declare (special pos-frac))
(declare (special tc-frac))
(declare (special tc-bit-vector))
(declare (special tc-exp))
(declare (special zero-flag))
(setq ieee-frac (get-bits  0 22 bv))
(setq ieee-exp  (get-bits 23 30 bv))
(setq ieee-sign (get-bits 31 31 bv))
(set-flags ieee-sign ieee-exp ieee-frac)    ;Set flags for special cases.

;; logical shift right one bit and add hidden bit

(setq pos-frac (set-bit 22 1 (right-shift 1 ieee-frac)))

;; For negative numbers, complement the shifted mantissa which puts the
;; mantissa into two's complement form. The mantissa is complemented by
;; inverting all bits in the bit-vector and then adding 1 to the inverted bv.
;; In two's complement form, bit 23 is needed because it is the sign bit.
;; Bit 23 is already zero in pos-frac. And for negative numbers it gets set
;; to one.

(setq tc-frac (if (= ieee-sign 0) pos-frac
                  (get-bits 0 23(+ (lognot pos-frac) 1))))

;; Subtract 127 from the ieee biased exponent, and
;; add 1 to the exponent for shifting the mantissa to the right
;; (since we shifted the mantissa from, say, 1.11 to 0.111).
;; (minus 127 plus 1 = -126)


(setq tc-exp (get-bits 0 7 (- ieee-exp 126)))

;combine tc-frac and tc-exp into tc-bit-vector

(setq tc-bit-vector (get-bits  0 23 tc-frac)) ;could just set it to tc-frac
(setq tc-bit-vector (put-bits tc-exp 24 31 tc-bit-vector))


;; The following code prints the results of the transformation on the screen,
;; along with various intermediate values.
```

```
(princ "                    MSB                    LSB")
(terpri)
(princ "The bit positions are   : ")
(princ "3322222222221111111111")
(terpri)
(princ "                    10987654321098765432109876543210")
(terpri)
(terpri)
(ieee-bv-to-dec ieee-sign ieee-exp ieee-frac)
(princ "The IEEE bit-vector is   : ")
(princ (format nil "~32,'0b"(logand #xffffffff bv)))
(terpri)
(princ "The IEEE fraction field is: ")
(princ (format nil "        ~23,'0b" ieee-frac))
(terpri)
(princ "The IEEE exponent field is: ")
(princ (format nil " ~8,'0b"ieee-exp))
(terpri)
(princ "The IEEE sign field is    : ")
(princ ieee-sign)
(terpri)
(princ "The positive fraction is : ")
(princ (format nil "       ~24,'0b" pos-frac))
(terpri)
(princ "The TC fraction field is : ")
(Princ (format nil "       ~24,'0b"  tc-frac))
(terpri)
(princ "The TC exponent field is : ")
(princ (format nil "~8,'0b"  tc-exp))
(terpri)
(princ "The TC bit-vector is     : ")
(princ (format nil "~32,'0b" (logand #xffffffff tc-bit-vector)))
(terpri)
(check-flags)                  ;if flags are set, print out override value
(tc-bv-to-dec tc-bit-vector)
(terpri))


;;; SET-FLAGS sets the appropriate flags out of the following seven flags:
;;;    1. Exponent overflow
;;;    2. Exponent underflow
;;;    3. Infinity
;;;    4. Unnormalized number or zero (exponent all zeroes)
;;;    5. Zero (exponent and fraction all zeroes)
;;;    6. lsb=1 to track loss of precision
;;;    7. Sign negative (bit 31 = 1)

;;; Overflow, infinity, and exp=0 are mutually exclusive.  If the exp=0,
;;; we go on to check if the fraction=0 or if we have underflow. The
;;; lsb=1-flag and sign-neg-flag could be set anytime, regardless of the
;;; other flags. The zero flag and underflow flag are mutually exclusive.
;;; If the fraction is zero, we exit the cond and don't set the
;;; exp-underflow-flag. The unnormalized-number-or-zero-flag and the
;;; zero-flag are not mutually exclusive. They can both be set. If the
;;; zero-flag is set, we know the unnormalized-number-or-zero-flag is set.


(defun set-flags (ieee-sign ieee-exp ieee-frac)
 (declare (special exp-overflow-flag))
 (declare (special exp-underflow-flag))
 (declare (special infinity-flag))
 (declare (special unnorm-or-zero-flag))
 (declare (special zero-flag))
 (declare (special lsb=1-flag))
 (declare (special sign-neg-flag))
 (setq exp-overflow-flag nil)
 (setq exp-underflow-flag nil)
 (setq infinity-flag nil)
 (setq unnorm-or-zero-flag nil)
 (setq zero-flag nil)
 (setq lsb=1-flag nil)
 (setq sign-neg-flag nil)
 (terpri)


 ;; The following cond checks for 3 mutually exclusive conditions:
 ;;    exp = +127 - overflow
 ;;    exp = +128 - infinity
 ;;    exp = zero

 ;; The largest number transformable is 1.11...1 * 2**126
 ;; (biased exponent = 253 = 11111101)
 ;; When we transform this number to tc, we get:  0.11...1 * 2**127
 ;; If the exponent equals +127 (biased exponent equals 254 = 11111110),
 ;; we get exponent overflow when we attempt the transformation.

 (cond ((= ieee-exp #b11111110)              ;exp = +127
    (setq  exp-overflow-flag t)
    (princ "Exponent overflow flag set.")
    (terpri))

    ;; If the exponent equals +128 we have infinity.
    ;; (biased exponent equals 255 = 11111111)

    ((= ieee-exp #b11111111)      ;exp = +128 which is infinity
     (setq  infinity-flag t)
     (princ "Infinity flag set.")
 (terpri))

 ;If it's an unnormalized number make sure it is not too small to
 ;transform.
; The smallest tc pos # is 0.5 * 2**-128.  This mantissa in binary
; is equal to  0.1 * 2**-128 = 0.001 * 2**-126
; Note: this last representation is illegal in tc since the
; mantissa is not normalized.
; The next smaller number (1too-small) is 0.0001111... * 2**-126.
; So if we have three zeros to the right of the decimal point
; (bits 22, 21, and 20) we know the positive fraction is too small
;  to be transformed.
; The negative case is slightly different than the positive case.
; The smallest tc neg # is -0.5000001 * 2**-128.  The magnitude of the
; mantissa has to
; be just above 1/2. This mantissa in binary sign-magnitude is equal to
;   0.100000...01 * 2**-128 = 0.0100000...001? * 2**-127
;                          = 0.00100000...0001? * 2**-126
; The next smaller number (neg-unnrm-1too-small) is
; -0.00100...00 * 2**-126 = -0.125 * 2**-126

((zerop ieee-exp) (setq unnorm-or-zero-flag t)
            (princ "Unnormalized-number-or-zero-flag set.")
         (terpri)
         (cond ((zerop ieee-frac)  ;if zero, set zero-flag
            (setq zero-flag t)
            (princ "Zero-flag set.")
            (terpri)
           )

           ; 0.000xxx... * 2**-126 is too small to be
           ; transformed, whether positive or negative.

           ((zerop (get-bits 20 22 ieee-frac))  ;0.0001
            (setq  exp-underflow-flag t)
            (princ "Exponent underflow flag set.")
            (terpri)
           )

           ;
           ((and                            ;-0.00100...0
             (= ieee-sign 1)

             ; The order of the bits makes this confusing.
             ; Actually, we don't need to check bit 20=1
             ;  since if bit 20 was 0 it would have been
             ;  caught by the previous conditon.

             (= (get-bits 20 22 ieee-frac) #b001)
```

```
                  (zerop (get-bits 0 19 ieee-frac)))
                 (setq  exp-underflow-flag t)
                 (princ "Exponent underflow flag set (neg sp).")
                 ; Negative smallest special case
                 (terpri)))))

    ;alternately      ((and
    ;                 (= ieee-sign 1)
      ;; #x800...00 = 1000...00 = 0.1000...00 = 1/2
    ;                 (<= ieee-frac #x80000)


 ;; If the sign bit is one, it's a negative number.
 ;; This works because the upper 30 bits in ieee-sign were set to zero by
 ;; GET-BITS.

 (cond ((= ieee-sign 1)
    (setq sign-neg-flag t)))

 ;; If lsb=1, we lose 1 bit of precision.  There's no way around
 ;; this loss of precision since the IEEE format has 1 more bit
 ;; of precision than the TC format.  However, this is not true
 ;; for unnormalized numbers.  Since they don't have a hidden
 ;; bit, we don't need to lose any precision (as long as they are
 ;; not too small to be transformed).

 (cond ((logbitp 0 ieee-frac)           ;If bit 0=1
    (setq lsb=1-flag t)
    (princ "lsb=1-flag set.")
    (terpri)
    (cond ((not  unnorm-or-zero-flag)
        (princ "One bit of precision lost.")
        (terpri)))))
)


;;; CHECK-FLAGS could  be done in software by the destination processor.
;;; It checks to see if any of the special case flags are set, and modifies the
;;; output of the hardware.  It prints out a flag override bit-vector,
;;; and updates the global variable tc-bit-vector.  It doesn't return anything
;;; in particular.  A more descriptive name might be CHECK-FLAGS-AND-
OVERRIDE.

(defun check-flags()                ;no input arg., uses global var.
 (declare (special exp-overflow-flag))
 (declare (special exp-underflow-flag))
 (declare (special infinity-flag))
```

```
(declare (special unnorm-or-zero-flag))
(declare (special zero-flag))
(declare (special lsb=1-flag))
(declare (special tc-bit-vector))
(declare (special tc-exp))
(declare (special tc-frac))
(declare (special sign-neg-flag))
(declare (special pos-zero))
(declare (special tc-pos-biggest))
(declare (special tc-pos-smallest))
(declare (special tc-neg-biggest))
(declare (special tc-neg-smallest))


;; If none of the flags are set, exit this function.
;; Don't bother checking the lsb=1-flag since there is no
;;   flag override bv for that case. We just lose one bit of precision.
;; Don't check the sign-neg-flag either since a negative number by itself
;;  is not a special case requiring an override. Although note that if one
;;  of the other five flags is set, the sign-neg-flag is sometimes used to
;;  determine which override bv to use.

(if (not
   (or exp-overflow-flag exp-underflow-flag infinity-flag
      unnorm-or-zero-flag zero-flag))
   (return-from check-flags nil))

;; Depending on what flag is set, make the appropriate
;; corrections. The following cond checks exponent overflow,
;; infinity, exponent underflow, zero, and unnorm-or-zero-flag.
;; The first four flags are mutually exclusive, but not the last
;; two. If the zero-flag is set, the unnorm-or-zero-flag is
;; also set. But if the zero flag is set, we don't care if the
;; unnorm-or-zero-flag is set.

(cond
 (zero-flag
  (setq tc-bit-vector pos-zero)              ;or set bits 22, 25, and 31 to zero
  )


 (exp-overflow-flag                ;If exponent overflow
  (if sign-neg-flag
   (setq tc-bit-vector tc-neg-biggest)
   (setq tc-bit-vector tc-pos-biggest)
   )
  )

 (exp-underflow-flag
 (if sign-neg-flag
  (setq tc-bit-vector tc-neg-smallest)
  (setq tc-bit-vector tc-pos-smallest)
  )
 )

(infinity-flag                  ;if plus or minus infinity

 (if sign-neg-flag
  (setq tc-bit-vector tc-neg-biggest) ;minus infinity
  (setq tc-bit-vector tc-pos-biggest) ;plus infinity
  )
 )

;; For unnormalized numbers only:
;; One bit of precision has been lost due to erroneously inserting
;; the hidden bit and then taking it back out (i.e. bit 0 is always 0).
;; Regain the lost bit with the lsb=1-flag.

;; We don't actually need to check that the zero-flag is not
;; set, because if the zero-flag had been set, we would have already
;; exited the cond.  But it is included for clarity.

((and unnorm-or-zero-flag
   (not zero-flag))

 ;; for negative unnormalized numbers, complement the mantissa to get it
 ;; back to the uncomplemented form.
 ;; There may be a way to do this without uncomplementing negative numbers,
 ;; but it isn't obvious.

 (when sign-neg-flag
  (setq tc-bit-vector (complement-bits 0 23 tc-bit-vector)))

 ;; For all unnormalized numbers:
 ;;  Since a hidden bit was erroneously inserted,left-shift bits 0-21.  In
 ;;  the shift, bit 21 overwrites the one that was erroneously inserted
 ;;  into bit 22.

 (setq tc-bit-vector
    (put-bits
     (left-shift 1 (get-bits 0 21 tc-bit-vector))
     0 22 tc-bit-vector)
    )

 ;;  Put back the lsb if there was one in the first place.

 (when lsb=1-flag
  (setq tc-bit-vector (set-bit 0 1 tc-bit-vector)))
  ;; for negative unnormalized numbers, complement the mantissa to get it
  ;; back to the complemented form.

  (when sign-neg-flag
   (setq tc-bit-vector (complement-bits 0 23 tc-bit-vector)))

  ;; Print the bit-vector with the hidden bit removed.

  (princ "Remove the hidden bit    : ")
  (princ (format nil "~32,'0b" (logand #xffffffff tc-bit-vector)))
  (terpri)

  ;; IEEE unnormalized numbers have all zeros in the exponent, which in IEEE
  ;; is considered to have a value of -126.  This usually works out pretty
  ;; well since IEEE-TO-TC subtracts 126 from the exponent so we get:
  ;;  (0-126 = -126).  The only
  ;; problem is, TC doesn't allow unnormalized mantissas.  There is a range
  ;; of unnormalized IEEE numbers with mantissas between 0.011...1 and
  ;; 0.0010...0 which are still large enough to be transformed into TC, but
  ;; which require normalization.  The mantissa needs to be left-shifted once
  ;; or twice and the exponent decremented by one or two.

  (when (equal (logbitp 23 tc-bit-vector)
        (logbitp 22 tc-bit-vector))
    (if (equal (logbitp 23 tc-bit-vector)
        (logbitp 21 tc-bit-vector))
      (progn (setq tc-exp (+ -2 (get-bits 24 31 tc-bit-vector)))
         (setq tc-frac (left-shift 2 (get-bits 0 23 tc-bit-vector))))
      (progn (setq tc-exp (+ -1 (get-bits 24 31 tc-bit-vector)))
         (setq tc-frac (left-shift 1 (get-bits 0 23 tc-bit-vector)))))

    (setq tc-bit-vector tc-frac)
    (setq tc-bit-vector (put-bits tc-exp 24 31 tc-bit-vector))
    )
 )  ; end of ((and unnorm-or-zero-flag (not zero-flag))
)              ;end of flag checking cond


;; Print the corrected (override) bit-vector.

(princ "The flag override bv is  : ")
(princ (format nil "~32,'0b" (logand #xffffffff tc-bit-vector)))
(terpri)
)


;;; IEEE-BV-TO-DEC prints out the value of the ieee bit-vector in decimal.
;;; It doesn't have anything to do with the transformation.
;;; &aux - declares local variables; used for recursive calls

(defun ieee-bv-to-dec(ieee-sign ieee-exp ieee-frac &aux sum exp)

 (declare (special infinity-flag))
 (princ "Value of IEEE bitvector is:  ")
 ;if sign-bit is one, print a minus sign
 (if (= ieee-sign 1) (princ "-") (princ " "))

 ;if infinity, print infinity and exit
 (when infinity-flag
   (princ "Infinity")
   (terpri)
   (return-from ieee-bv-to-dec nil))

 (if (zerop ieee-exp)

     ;; IF the exp bits are all zero, the number is unnormalized or zero.
     ;; In either case, there is no hidden bit.

     (progn (setq sum (sum-frac ieee-frac 22 -1 0))
          (if (= sum 0)

             ;IF mantissa bits are all zero, the exp=0

             (setq exp 0)

             ;ELSE the number is unnormalized and the exp=-126 (by def.)

             (setq exp -126)))

     ;; ELSE add 1.0 to the mantissa for the hidden bit and
     ;; subtract 127 from the biased exponent to get the actual exponent.

     (progn (setq sum (+ 1 (sum-frac ieee-frac 22 -1 0)))
          (setq exp (- ieee-exp 127))))
 (princ (format nil "~f" sum))
 (princ " * 2**")
 (princ exp)
 (terpri)
)


;;; SUM-FRAC adds up the weighted values of the bits in the fraction.
;;; This is a recursive function.
;;; The initial call has:
;;;    bit-pos = the number of bits in the fraction,
;;;    weight = -1  (i.e. 2**-1 = 1/2)
;;;   sum = 0.

(defun sum-frac (bit-vector bit-pos weight sum)
 (if (logbitp bit-pos bit-vector)          ;If the bit = 1

    ;add the weighted value of that bit to the sum.

    (setq sum (+ sum (expt 2 weight)))
    nil)                         ;otherwise don't add anything to the sum

 ;Base case of recursion: after adding the lsb(i.e. bit 0),
 (if (= bit-pos 0)
    sum              ;return the sum.

    ;ELSE recursive call with the next bit position and decrement the weight.

    (sum-frac bit-vector(- bit-pos 1) (- weight 1) sum)))

;;; TC-BV-TO-DEC prints out the value of the two's complement bit-vector in
;;; decimal. It doesn't have anything to do with the transformation.
;;; &aux - declares local variables; used for recursion

(defun tc-bv-to-dec(tc-bit-vector &aux sum exp)
 (declare (special unnorm-or-zero-flag))
 (declare (special infinity-flag))

 (princ "Value of TC bitvector is  :  ")

 ;; if mantissa sign-bit is one (i.e. negative), add -1 to fraction

 (if (logbitp 23 tc-bit-vector)
    (setq sum (+ -1 (sum-frac tc-bit-vector 22 -1 0)))
    (progn (setq sum (sum-frac tc-bit-vector 22 -1 0))
       (princ " ")))

 ;; If exponent sign bit (bit 31) is 1 (i.e.  negative exponent), add -128 to
 ;; the exponent.  For positive exponents, LISP can figure out the decimal
 ;; value of the bit vector since it is an integer.

 (if (logbitp 31 tc-bit-vector)
    (setq exp (+ -128 (get-bits 24 30 tc-bit-vector)))
    (setq exp (get-bits 24 30 tc-bit-vector)))

 (princ (format nil "~f" sum))
 (princ " * 2**")
 (princ exp)
 (terpri)

 ;; The following makes it easier to compare the ieee and tc values.
;; It is used for unnormalized numbers with a tc-exp of -127 or -128.

 (when unnorm-or-zero-flag
  (when (= exp -127)
    (princ "Equiv. value of TC bv is  :  ")
    (if (not (logbitp 23 tc-bit-vector))
      (princ " "))
    (princ (format nil "~f" (/ sum 2)))
    (princ " * 2**")
    (princ (+ 1 exp))
    (terpri))
  (when (= exp -128)
    (princ "Equiv. value of TC bv is  :  ")
    (if (not (logbitp 23 tc-bit-vector))
      (princ " "))
    (princ (format nil "~f" (/ sum 4)))
    (princ " * 2**")
    (princ (+ 2 exp))
    (terpri)))

 ;; The following makes it easier to compare the ieee and tc values.
 ;; It is used for all but unnormalized numbers or infinity.
 ;; Shift the fraction left one bit (multiply by two),
 ;; and subtract one from the exponent.
 ;; Don't do the following for unnormalized numbers or zero.

 (if (or  infinity-flag unnorm-or-zero-flag)
    (return-from tc-bv-to-dec nil))
 (princ "Equiv. value of TC bv is  :  ")
 (if (not (logbitp 23 tc-bit-vector))  ;print a space before pos. #s
    (princ " "))
 (princ (format nil "~f" (* 2 sum)))
 (princ " * 2**")
 (princ (- exp 1))
 (terpri)
)


;;; Misc. test values
;;; The following numbers are  represented in the IEEE format.
;;; If the suffix "-x" or the phrase "too" appears in the name, that means
;;; the name describes the significance of the number in regards to the
;;; transformation.  For example, pos-biggest-x is the largest positive number
;;; that can be correctly transformed, and pos-1too-big is a positive number
;;; that is one too big to be correctly transformed.  Otherwise, the name
;;; describes the significance of the number in the IEEE format (i.e.
;;; pos-infinity).

(setq pos-infinity        #b01111111101010101010101010101011)
```

```
; plus infinity - mantissa values don't matter

(setq pos-biggest       #b01111111011111111111111111111111)
; 1.9999999 * 2**127

(setq pos-1too-big      #b01111111100000000000000000000000)
; 1.0 * 2**127
; This is 1 larger than the largest positive ieee # that can be transformed

(setq pos-biggest-x     #b01111110111111111111111111111111)
; 1.9999999 * 2**126
; This is the largest positive ieee # that can be transformed

(setq pos-exp-pos       #b01000000111000000000000000000000) ; 1.75 * 2**2

(setq pos-exp-zero      #b00111111111000000000000000000000)     ; 1.75 * 2**0

(setq pos-exp-neg       #b00111110111000000000000000000000)     ; 1.75 * 2**-
2

(setq pos-smallest      #b00000000100000000000000000000000)     ; 1.0  * 2**-
126

(setq pos-unnorm-biggest  #b00000000011111111111111111111111)
; 0.9999999  * 2**-126
; This number happens to be H/W transformed correctly since the mantissa is all
;  1's.  We erroneously right-shift the mantissa by 1 and add the hidden bit
;  and get the same number we started with.  (The exponent for unnormalized
;  numbers is transformed correctly.)

(setq pos-unnorm        #b00000000010000000000000000000001)
; 0.50000012  * 2**-126

(setq pos-unnorm-norm-not-reqd-smallest
                  #b00000000010000000000000000000000)
; 0.5  * 2**-126

(setq pos-unnorm-norm-reqd-biggest
                  #b00000000001111111111111111111111)
; 0.4999999??  * 2**-126

(setq  pos-unnorm-norm-reqd
                  #b00000000001000000000000000000001)
; 0.25000012  * 2**-126

(setq pos-unnorm-smallest-x
                  #b00000000000100000000000000000000)
; 0.125  * 2**-126  - normalization required
```

```
(setq pos-unnrm-1too-small #b00000000000111111111111111111111)
; 0.12499988  * 2**-126

(setq pos-zero         #b00000000000000000000000000000000)  ; +0 * 2**0

(setq neg-zero         #b10000000000000000000000000000000)  ; -0 * 2**0

(setq neg-unnrm-1too-small
                 #b10000000000100000000000000000000)
;-0.1250...0  * 2**-126

(setq neg-unnorm-smallest-x
                 #b10000000000100000000000000000001)
;-0.12500012  * 2**-126

(setq neg-unnorm-norm-reqd
                 #b10000000001000000000000000000001)
;-0.25000012  * 2**-126

(setq neg-unnorm-norm-reqd-biggest-1
                 #b10000000001111111111111111111111)
;-0.4999999??  * 2**-126

(setq neg-unnorm-norm-reqd-biggest
                 #b10000000010000000000000000000000)
;-0.5  * 2**-126

(setq neg-unnorm-norm-not-reqd-smallest
                 #b10000000010000000000000000000001)
;-0.50000?1  * 2**-126

(setq neg-unnorm-biggest  #b10000000011111111111111111111111)
;-0.9999999  * 2**-126

(setq neg-smallest       #b10000000100000000000000000000000)      ;-1.0  * 2**-
126

(setq neg-exp-neg        #b10111110111000000000000000000000)      ;-1.75 * 2**-
2

(setq neg-exp-zero       #b10111111111000000000000000000000)      ;-1.75 * 2**0

(setq neg-exp-pos        #b11000000111000000000000000000000)  ;-1.75 * 2**2

(setq neg-biggest-x      #b11111111000000000000000000000000) ;-1.0 * 2**127
; This is the largest negative ieee # that can be transformed

(setq neg-1too-big       #b11111111000000000000000000000001)
;-1.00000012 * 2**127
```

```
; This is 1 larger than the largest negative ieee # that can be transformed

(setq neg-biggest       #b11111111011111111111111111111111)
;-1.9999999 * 2**127
; This is the largest negative ieee #

(setq neg-infinity      #b11111111101010101010101010101011)
; minus infinity - mantissa values don't matter



;;; The following names refer to the value of these bit vectors in the
;;;  two's complement representation.
;;; The two's complement representation requires that all mantissas be
;;; normalized.  This means that for positive numbers, bit 22 must be a one,
;;; and for negative numbers, bit 22 must be a zero.
;;; I don't understand why this is required.  There must be some mathematical
;;; reason for this requirement.

(setq tc-pos-biggest     #b01111111011111111111111111111111)
; 0.9999999 * 2**127

(setq tc-pos-smallest    #b10000000010000000000000000000000)
; 0.5  * 2**-128 = 0.1 * 2**-128 = 0.001 * 2**-126

(setq tc-neg-smallest    #b10000000101111111111111111111111)
; -0.5000001 * 2**-128

(setq tc-neg-smallesterxx  #b10000000110000000000000000000000)
; -0.5 * 2**-128  ILLEGAL - NOT NORMALIZED

(setq tc-neg-smallestestxx #b10000000111111111111111111111111)
; -0.00000011920929 * 2**-128  ILLEGAL - NOT NORMALIZED

(setq tc-neg-biggest     #b01111111110000000000000000000000)     ; -1.0 *
2**127


;;;EXAMPLES-POS performs IEEE-TO-TC on the positive test values above

(defun examples-pos ()
  (declare (special pos-infinity))
  (declare (special pos-biggest))
  (declare (special pos-1too-big))
  (declare (special pos-biggest-x))
  (declare (special pos-exp-pos))
  (declare (special pos-exp-zero))
  (declare (special pos-exp-neg))
  (declare (special pos-smallest))
```

```
(declare (special pos-unnorm-biggest))
(declare (special pos-unnorm))
(declare (special pos-unnorm-norm-not-reqd-smallest))
(declare (special pos-unnorm-norm-reqd-biggest))
(declare (special pos-unnorm-norm-reqd))
(declare (special pos-unnorm-smallest-x))
(declare (special pos-unnrm-1too-small))
(declare (special pos-zero))
(terpri)
(terpri)
(terpri)
(princ "pos-infinity")
(terpri)
(ieee-to-tc pos-infinity)
(terpri)
(princ "pos-biggest")
(terpri)
(ieee-to-tc pos-biggest)
(terpri)
(princ "pos-1too-big")
(terpri)
(ieee-to-tc pos-1too-big)
(terpri)
(princ "pos-biggest-x")
(terpri)
(ieee-to-tc pos-biggest-x)
(terpri)
(princ "pos-exp-pos")
(terpri)
(ieee-to-tc pos-exp-pos)
(terpri)
(princ "pos-exp-zero")
(terpri)
(ieee-to-tc pos-exp-zero)
(terpri)
(princ "pos-exp-neg")
(terpri)
(ieee-to-tc pos-exp-neg)
(terpri)
(princ "pos-smallest")
(terpri)
(ieee-to-tc pos-smallest)
(terpri)
(princ "pos-unnorm-biggest")
(terpri)
(ieee-to-tc pos-unnorm-biggest)
(terpri)
(princ "pos-unnorm")
(terpri)
```

```
 (ieee-to-tc pos-unnorm)
 (terpri)
 (princ " pos-unnorm-norm-not-reqd-smallest")
 (terpri)
 (ieee-to-tc  pos-unnorm-norm-not-reqd-smallest)
 (terpri)
 (princ "pos-unnorm-norm-reqd-biggest")
 (terpri)
 (ieee-to-tc pos-unnorm-norm-reqd-biggest)
 (terpri)
 (princ "pos-unnorm-norm-reqd")
 (terpri)
 (ieee-to-tc pos-unnorm-norm-reqd)
 (terpri)
 (princ "pos-unnorm-smallest-x")
 (terpri)
 (ieee-to-tc pos-unnorm-smallest-x)
 (terpri)
 (princ "pos-unnrm-1too-small")
 (terpri)
 (ieee-to-tc pos-unnrm-1too-small)
 (terpri)
 (princ "pos-zero")
 (terpri)
 (ieee-to-tc pos-zero)
 (terpri)
)

;;;EXAMPLES-NEG performs IEEE-TO-TC on the negative test values above

(defun examples-neg ()
 (declare (special neg-infinity))
 (declare (special neg-biggest))
 (declare (special neg-1too-big))
 (declare (special neg-biggest-x))
 (declare (special neg-exp pos))
 (declare (special neg-exp-zero))
 (declare (special neg-exp-neg))
 (declare (special neg-smallest))
 (declare (special neg-unnorm-biggest))
 (declare (special neg-unnorm))
 (declare (special neg-unnorm-norm-reqd))
 (declare (special neg-unnorm-norm-reqd-biggest))
 (declare (special neg-unnorm-norm-not-reqd?-smallest))
 (declare (special neg-unnorm-norm-not-reqd-smallest))
 (declare (special neg-unnorm-smallest-x))
 (declare (special neg-unnrm-1too-small))
 (declare (special neg-zero))
 (terpri)
```

```
(terpri)
(terpri)
(princ "neg-infinity")
(terpri)
(ieee-to-tc neg-infinity)
(terpri)
(princ "neg-biggest")
(terpri)
(ieee-to-tc neg-biggest)
(terpri)
(princ "neg-1too-big")
(terpri)
(ieee-to-tc neg-1too-big)
(terpri)
(princ "neg-biggest-x")
(terpri)
(ieee-to-tc neg-biggest-x)
(terpri)
(princ "neg-exp-pos")
(terpri)
(ieee-to-tc neg-exp-pos)
(terpri)
(princ "neg-exp-zero")
(terpri)
(ieee-to-tc neg-exp-zero)
(terpri)
(princ "neg-exp-neg")
(terpri)
(ieee-to-tc neg-exp-neg)
(terpri)
(princ "neg-smallest")
(terpri)
(ieee-to-tc neg-smallest)
(terpri)
(princ "neg-unnorm-biggest")
(terpri)
(ieee-to-tc neg-unnorm-biggest)
(terpri)
(princ "neg-unnorm-norm-not-reqd-smallest")
(terpri)
(ieee-to-tc neg-unnorm-norm-not-reqd-smallest)
(terpri)
(princ "neg-unnorm-norm-reqd-biggest")
(terpri)
(ieee-to-tc neg-unnorm-norm-reqd-biggest)
(terpri)
(princ "neg-unnorm-norm-reqd-biggest-1")
(terpri)
(ieee-to-tc neg-unnorm-norm-reqd-biggest-1)
```

```
 (terpri)
 (princ "neg-unnorm-norm-reqd")
 (terpri)
 (ieee-to-tc neg-unnorm-norm-reqd )
 (terpri)
 (princ "neg-unnorm-smallest-x")
 (terpri)
 (ieee-to-tc neg-unnorm-smallest-x)
 (terpri)
 (princ "neg-unnrm-1too-small")
 (terpri)
 (ieee-to-tc neg-unnrm-1too-small)
 (terpri)
 (princ "neg-zero")
 (terpri)
 (ieee-to-tc neg-zero)
 (terpri)
)
```

Appendix B
Queueing Models

This appendix lists the queueing models which were used to determine the queueing time (t_q) for the performance models (see §5.2.3.). The values are shown first, followed by the formulas which are used to calculate those values. The formula for any value cell can be determined by using the row and column of the value cell to look up the formula in the tables which follow. For example, the expected service time (Es) for Design A is in row 4, column AN, and has a value of 6.0 (Table B-1). The formula for this is shown in Table B-2 in row 4, column AN, and is "=tnet+tm". In Table B-1, rows 16 to 21 are used to compute the value of p_0. The formula for p_0 is given in §5.2.3..

Values

Table B-1: Values for queueing models

	AM	AN	AO	AP	AQ	AR	AS	AT	AU	AV	AW
1			Queueing Models - one for each design								
2											
3		A	B	CC	D	E	F	G	H	J	K
4	Es	6.0	6.0	7.9	6.0	11.3	11.3	11.3	11.3	11.3	11.3
5	Et	0.9	18.8	6.4	18.8	8.1	47.1	47.1	47.1	42.0	47.1
6	p0	0.0	0.1	0.0	0.1	0.0	0.2	0.2	0.2	0.2	0.2
7	W	29.1	15.3	33.1	15.3	48.2	24.6	24.6	24.6	26.2	24.6
8	tq	23.1	9.3	25.2	9.3	36.9	13.4	13.4	13.4	14.9	13.4
9											
10	Misc. calculations			0.2							
11				15.0		15.0					
12				28.1		28.1					
13				18.8		47.1					
14											
15	i										
16	0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
17	1	32.3	1.6	6.1	1.6	6.9	1.2	1.2	1.2	1.3	1.2
18	2	832.5	2.0	30.2	2.0	38.6	1.1	1.1	1.1	1.4	1.1
19	3	16112.2	1.9	111.1	1.9	160.8	0.8	0.8	0.8	1.2	0.8
20	4	207900.0	1.2	272.7	1.2	446.6	0.4	0.4	0.4	0.6	0.4
21	5	1341290.2	0.4	334.8	0.4	620.3	0.1	0.1	0.1	0.2	0.1

Formulas

Table B-2: Formulas for queueing models

	AN	AO
3	A	B
4	=tnet+tm	=tnet+tm
5	=1/(1/CPI+1/CPD)	=CPD/psm
6	=1/SUM(AN16:AN36)	=1/SUM(AO16:AO36)
7	=(N*AN4)/(1-AN6)-AN5	=(N*AO4)/(1-AO6)-AO5
8	=AN7-AN4	=AO7-AO4

Table B-3: Formulas for queueing models (cont.)

	AP
3	CC
4	=tnet+(psm/AP10)*tm+((1-psm)*pcm*(1+pdL)/AP10)*tmL
5	=1/(1/AP11+1/AP12+1/AP13)
6	=1/SUM(AP16:AP36)
7	=(N*AP4)/(1-AP6)-AP5
8	=AP7-AP4
9	
10	=psm+(1-psm)*pcm
11	=(CPI/pcm)
12	=(CPD/((1-psm)*pcm))
13	=CPD/psm

Table B-4: Formulas for queueing models (cont.)

	AQ	AR
3	D	E
4	=tnet+tm	=AS4
5	=CPD/psm	=1/(1/AR11+1/AR12+1/AR13)
6	=1/SUM(AQ16:AQ36)	=1/SUM(AR16:AR36)
7	=(N*AQ4)/(1-AQ6)-AQ5	=(N*AR4)/(1-AR6)-AR5
8	=AQ7-AQ4	=AR7-AR4
9		
10		
11		=(CPI/pcm)
12		=(CPD/((1-psm)*pcm))
13		=CPD/(psm*pcms)

Table B-5: Formulas for queueing models (cont.)

	AS	AT
3	F	G
4	=(1+pdL)*(tnet+tmL)	=AS4
5	=CPD/(psm*pcms)	=AS5
6	=1/SUM(AS16:AS36)	=1/SUM(AT16:AT36)
7	=(N*AS4)/(1-AS6)-AS5	=(N*AT4)/(1-AT6)-AT5
8	=AS7-AS4	=AT7-AT4

Table B-6: Formulas for queueing models (cont.)

	AU	AV
3	H	J
4	=AS4	=AS4
5	=AS5	=CPD/(pr*psm*pcms+pw*psm)
6	=1/SUM(AU16:AU36)	=1/SUM(AV16:AV36)
7	=(N*AU4)/(1-AU6)-AU5	=(N*AV4)/(1-AV6)-AV5
8	=AU7-AU4	=AV7-AV4

Table B-7: Formulas for queueing models (cont.)

	AW
3	K
4	=AS4
5	=AS5
6	=1/SUM(AW16:AW36)
7	=(N*AW4)/(1-AW6)-AW5
8	=AW7-AW4

Bibliography

[Alle80] Allen, A.O., "Queueing Models of Computer Systems", *Computer*, April 1980.

[AMD89] *Am29000 Users Manual*, Advanced Micro Devices, Inc., 1988.

[ArBa86] Archibald, J. and J.L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, Vol. 4, No. 4, November 1986, pp. 273-298.

[Arno89] Arnould, E.A., F.J. Bitz, E.C. Cooper, H.T. Kung, R.D. Sansom, and P.A. Steenkiste, "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers", *3rd International Conference on Architectural Support for Programming Languages and Operating Systems* , April 1989, pp. 205-216.

[ATF89] Advanced Tactical Fighter System Program Office, Joint Avionics Division, ASD/YFAJ, Wright-Patterson Air Force Base, OH Telecon, October 25, 1989.

[BaBr89] Baylor, S.J. and F.A. Briggs, "The Effects of Cache Coherency on the Performance of Parallel PDE Algorithms in Multiprocessors", *Proceedings 1989 International Conference on Parallel Processing*, 1989, pp. I-233 - I-236.

[BiDe86] Bitar, B. and A.M. Despain, "Multiprocessor Cache Synchronization", *Proceedings 13th International Symposium on Computer Architecture,* 1986, pp. 424-433.

[BiFo87] Bisiani, R. and A. Forin, "Architectural Support for Multilanguage Parallel Programming on Heterogeneous Systems", 1987, ACM

[Bisi87] Bisiani, R., *Heterogeneous Parallel Processing: The Agora Shared Memory*, CMU-CS-87-112, Carnegie-Mellon University, Computer Science Department, March 1987.

[BlBr87] Blaauw, G.A. and F.P. Brooks, Jr., *Computer Architecture, Volume 1 - Design Decisions*, Fall 1987, Draft.

[Borc86] Borchardt, G.C., "STAR: A Computer Language for Hybrid AI Applications", *Coupling Symbolic and Numerical Computing in Expert Systems*, Elsevier Science Publishers, 1986.

[Caro86] Carothers, J.D. , *Adjacency-Matrix Reduction Algorithms*, M.S. Thesis, Dept. of Electrical and Computer Engineering, University of Texas at Austin, 1986.

[Chan90] Chan, T.Y., Design Engineer TMS34010, Texas Instruments, Personal Communication, February 2, 1990.

[Cher83] Cheriton, D.R. and W. Zwaenepoel, "The Distributed V Kernel and Its Performance for Diskless Workstations", *Proceedings 9th ACM Symposium on Operating System Principles*, October 1983.

[Cohe81] Cohen, Danny, "On Holy Wars and a Plea for Peace", *Computer*, October 1981.

[CrWa89] Cragon, H.G. and W.J. Watson, "The TI Advanced Scientific Computer", *Computer*, January 1989.

[Craw90] Crawford, J.H., "The i486 CPU: Executing Instructions in One Clock Cycle", *IEEE Micro*, February 1990.

[Dasg89] Dasgupta, S. *Computer Architecture: A Modern Synthesis, Volume 1: Foundations*, John Wiley & Sons, 1989.

[Dean87] Dean, M.A., R.M. Sands, and R.E. Schantz, "Canonical Data Representation in the Cronus Distributed Operating System", *Proceedings IEEE Infocom 87*, San Francisco, CA, March 31-April 2, 1987.

[DePa85] Despain, A.M. and Y.N. Patt, "Aquarius - A High Performance Computing System for Symbolic/Numeric Applications", *COMPCON 1985*, IEEE Computer Conference.

[EgKa88] Eggers, S.J. and R.H. Katz, "The Effect of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation", *Proceedings 15th International Symposium on Computer Architecture.* 1988, pp. 373-382.

[EgKa89] Eggers, S.J., and Katz, R.H. "A Characterization of Sharing on the Cache and Bus Performance of Parallel Programs", *Proceedings 3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1989, pp. 257-270.

[ExLX86] *Explorer LX Communications Guide*, Texas Instruments, Inc., Preliminary draft, September 19, 1986.

[ExLX87] *Explorer LX Users Guide*, Texas Instruments, Inc., Revision A, July 1987.

[Ferr78] Ferrari, D. *Computer Systems Performance Evaluation*, Prentice-Hall, 1978.

[Frie88] Friedman, G.M., The MITRE Corp., (coauthor of [LAN 87]), Personal communication, June 15, 1988.

[Gill87] Gill, J.L., *Processor Performance as a Function of Memory Bandwidth*, M.S. Thesis, University of Texas at Austin, May 1987.

[Glas90] Glass, L.B., "Math Coprocessors", *Byte*,Vol. 15, No. 1, January 1990.

[Harr87] *Harris C Reference Manual*, Harris Corporation, April 1987.

[Heil87] Heilmeier, G., Senior Vice-President and Chief Technical Officer, Texas Instruments Computer Science Center, Personal Communication, June 23, 1987.

[Hell67] Hellerman, H., *Digital Computer System Principles*, McGraw-Hill Book Company, 1967.

[Hwan79] Hwang, K., *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley and Sons, Inc. 1979.

[HwBr84] Hwang, K. and F.A..Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc., 1984.

[i486 89] *Intel i486 Microprocessor*, Intel Corporation, 1989.

[i486 90] *Intel i486 Microprocessor Programmer's Reference Manual*, Intel Corporation, 1990.

[i960 89] *Intel Solutions 960*, Intel Corporation, 1989.

[IEEE85] IEEE Standard for Binary Floating Point Arithmetic, ANSI IEEE 754 1985.

[JIAW88] Joint Integrated Avionics Working Group, High Speed Data Bus Interface Module Standard, draft 13 Dec 88, ASD/YFAJ, Wright-Patterson Air Force Base, OH.

[JIAW89] Joint Integrated Avionics Working Group, PI[Parallel Interface]-Bus Specification, draft 23 Sep 89, ASD/YFAJ, Wright-Patterson Air Force Base, OH.

[Kirr83] Kirrman, H. "Data Format and Bus Compatibility in Multiprocessors", *IEEE Micro*, Vol. 3, No. 4, August 1983.

[Klei76] Kleinrock, L. *Queueing Systems, Volume 2: Computer Applications*, John Wiley & Sons, 1976.

[LAN 87] *LAN Protocol Translating Gateways: High-Performance Solutions Using VLSI Technology*, The MITRE Corp., M87-11, March 1987.

[LiMa87] Lipovski, G.J. and M. Malek, *Parallel Computing: Theory and Comparisons*, John Wiley & Sons, 1987.

[M88K89] *MC88100 RISC Microprocessor Users Manual*, Motorola, Inc., Second Edition, 1989.

[M688 87] *MC68881/MC68882 Floating Point Coprocessor Users Manual*, Motorola, Inc., First Edition, 1987.

[Matu67] Matula, D.W., "Base Conversion Mappings", *Proceedings AFIPS 1967 Spring Joint Computer Conference*, Vol. 30, pp. 311-318.

[Matu68] Matula, D.W. "In-and-Out Conversions", *Communications of the ACM*, January 1968, Vol. 11, No. 1, pp. 47-50.

[Matu70] Matula, D.W. "A Formalization of Floating-Point Numeric Base Conversion", *IEEE Transactions on Computers*, August 1970, Vol. C-19, No. 8, pp. 681-692.

[MiBa89] Min, S.L. and J.L. Baer, "A Timestamp-based Cache Coherence Scheme", *Proceedings 1989 International Conference on Parallel Processing*, 1989, pp. I-23 - I-32.

[MILS82] MIL-STD-1750A (USAF), Military Standard Sixteen-Bit Computer Instruction Set Architecture, 2 July 1980, revised 21 May 1982.

[MIPS88] Kane, G., *MIPS RISC Architecture*, MIPS Computer Systems, Inc., Prentice-Hall, Inc., 1988.

[Morr86] Morris, L.R., "Digital Signal Processing Microprocessors: Forward to the Past?", *IEEE Micro*, Vol. 6, No. 6, December 1986, pp. 6-7.

[Morr88] Morris, L.R. and S.A. Dyer, "Floating-Point Digital Signal Processing Chips, A New Era for DSP Systems Design", *IEEE Micro*, Vol. 8, No. 6, December 1988, pp. 10-12.

[Notk87] Notkin, D., et al, "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity", *Communications of the ACM*, February 1987.

[Rash87] Rashid, R., "Mach: A New Foundation for Multiprocessor Systems Development", *COMPCON Spring 87*, IEEE Computer Society, 1987.

[Rash88] Rashid, R.F., et al, "Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *IEEE Transactions on Computers*, C-37(8):896-908, August 1988.

[Reif90] Reifel, Mitch, Texas Instruments, Inc., Telecom, March 15, 1990.

[Rest89] R. Restle, *TMS320C30 - IEEE Floating Point Format Converter*, Texas Instruments Design Report, P.O. Box 1443, MS 737, Houston, TX 77251-1443, November 1989.

[Rose90] Rose, M.T., *The Open Book: A Practical Perspective on OSI*, Prentice-Hall, 1990.

[Scha88] Schantz, R.E., BBN Laboratories, Telecom, July 5, 1988.

[Serr90] Serrano, C., "88000 Presentation", Motorola University Symposium, Austin, TX, June 4-5, 1990.

[Smit85] Smith,A.J., "Cache Evaluation and the Impact of Workload Choice", *Proceedings 12th International Symposium on Computer Architecture*, 1985, pp. 64-73.

[StCr88] Strevell, M.W., and H.G. Cragon, "High-Speed Transformation of Primitive Data Types in a Heterogeneous Distributed Computer System", *Proceedings 8th International Conference on Distributed Computing Systems*, San Jose, CA, June 1988.

[SwSm86] Sweazy, P. and A.J. Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus", *Proceedings 13th International Symposium on Computer Architecture*. 1986, pp. 414-423.

[TMS88] *TMS320 Third-Generation User's Guide*, Texas Instruments, Inc., 1988.

[VAX86] *VAX Architecture Handbook*, Digital Equipment Corporation, 1986.

[Wilk65] Wilkes, M.V., "Slave Memories and Dynamic Storage Allocation", *IEEE Transactions on Electronic Computers*, Vol EC-14, No. 2, April 1965. pp. 270-271.

[WiMa89] Wittie, L. and C. Maples, "Merlin: Massively Parallel Heterogeneous Computing", *Proceedings 1989 International Conference on Parallel Processing*, August 1989, pp. I-142 - I-150.

[XDR86] XDR, External Data Representation Protocol Specification, *Networking on the Sun Workstation*, Sun Microsystems, February 1986.

VITA

Michael W. Strevell was born in Albany, New York, on April 23, 1955, the son of Jane Marie Strevell and John William Strevell. After graduation from Carroll High School, Dayton, Ohio, in 1973, he entered the United States Air Force Academy in Colorado Springs, Colorado. He graduated from the Air Force Academy in 1977 and received the degree of Bachelor of Science in Electrical Engineering and a regular commission as a second lieutenant in the United States Air Force. In 1978 he completed pilot training at Williams Air Force Base, Phoenix, Arizona. From 1978 to 1983 he flew the B-52H Intercontinental Nuclear Bomber at Minot Air Force Base, North Dakota. During that time he also attended graduate school at North Dakota State University and in 1983 he received the degree of Master of Science in Industrial Engineering and Management. From 1983 to 1985 he was a project manager in Computer Integrated Manufacturing, and then Deputy Director of Artificial Intelligence Concepts in the Manufacturing Technology Division of the Materials Laboratory, Wright-Patterson Air Force Base, Dayton, Ohio. In January 1986 he entered the Graduate School of The University of Texas at Austin, and in December 1986 he received the degree of Master of Science in Electrical Engineering. During 1987 he completed his course work for the degree of Doctor of Philosophy. In January 1988 he joined the faculty of the United States Air Force Academy and taught in the Department of Electrical Engineering. He is a registered professional engineer in the state of Colorado. In June 1989 he returned to The University of Texas to complete his research for the degree of Doctor of Philosophy.

Permanent address: 230 Terrance Drive
Naperville, Illinois 60565

This dissertation was typed by the author.